Surveillance comportementale de systèmes et logiciels embarqués par signature disjointe

Selma Bergaoui

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Nanoélectronique et Nanotechnologie**

Arrêté ministériel : 7 août 2006

Présentée par

Salma BERGAOUI

Thèse dirigée par **Régis LEVEUGLE**

Préparée au sein du **Laboratoire TIMA**
dans **l'École Doctorale EEATS**

Surveillance comportementale de systèmes et logiciels embarqués par signature disjointe

Thèse soutenue publiquement le **6 Juin 2013**,
devant le jury composé de :

Mme. Lorena Anghel
Professeur, Université de Grenoble, Grenoble-INP (Président)
M. Matteo Sonza-Reorda
Professeur, Politecnico de Turin (Rapporteur)
M. Philippe Coussy
MCF DHDR, Université de Bretagne Sud (Rapporteur)
M. Régis Leveugle
Professeur, Université de Grenoble, Grenoble-INP (Directeur de thèse)
M. Yannick Teglia
STMicroelectronics (Examinateur)

À Bibi, mes parents, ma sœur et mon petit frère
À Gradou, Momo et Kattous

Remerciements

Ces travaux de thèse se sont déroulés au sein du laboratoire TIMA de Grenoble, dont je tiens à remercier en particulier la directrice, Dominique Borrione, ainsi que tout le personnel administratif. Je pense notamment à Anne-Laure Fourneret-Itié, Sophie Martineau, Marie Christine Salizzoni, Laurence Ben Tito et Youness Rajab ; sans oublier, Ahmed Khalid, Frédéric Chevrot et Nicolas Garnier du service info.

Je tiens à exprimer ma profonde gratitude aux membres du jury, en commençant par Lorena Anghel, la présidente du collège doctoral de Grenoble INP, en sa qualité de présidente de mon jury. Je tiens également à remercier les rapporteurs de ce jury, Matteo Sonza Reorda, responsable du groupe CAD au Politecnico di Torino, et Philippe Coussy, maître de conférence à l'UBS. Un grand merci également à Yannick Teglia, qui fût mon examinateur lors de la soutenance.

Je tiens aussi à adresser mes plus sincères remerciements à mon directeur de thèse, Régis Leveugle pour sa patience, son soutien, et ses conseils. Vous êtes le meilleur chef au monde!

Pour sa contribution à ces travaux, je tiens à remercier chaleureusement Pierre Vanhauwaert qui a passé ses matinées "à modifier le watchdog" (La bise à Solenne et à Camille !).

J'ai également une pensée pour Alexandre Chagoya et Robin Rolland Girot, les maîtres du CIME. J'en profite pour remercier Katell Morin Allory, François Cayre, et Laurent Fesquet, avec qui j'ai eu la chance d'animer certains TDs et projets à Phelma.

Cette thèse n'aurait pu être possible sans le support de plusieurs personnes qui m'ont entourée et encouragée. Je remercie tout particulièrement, Jean-Baptiste (et Marion) Ferron, Paolo (et Anna) Maistri, Diego (et Gabriella) Alberto Da Foglizzo, Maryam Bahmani, Giota Papavramidou, Fabien Chaix, Adrien Prost Boucle, Vladimir Pasca, Wassim Mansour, Michael Dimopolous et Negin Javaheri, qui ont partagé avec moi, ces quatre années au laboratoire.

Je veux également dire merci à Mihail Nicolaïdis, Raul Velazco, Nacer Zergainoh et Skandar Basrour pour leurs conseils durant toute ma thèse.

Enfin, je remercie énormément ma famille: Mamati, Papati, Tata Saloua, Tonton Anouar, Meriem, Mohamed Amine, Azza, Nabil et bien évidemment mon mari, Mohamed Ben Jrad, pour leur soutien et leurs encouragements durant toutes ces années.

Table des matières

Liste des figures

Liste des tableaux

Annexe -A-: Présentation du jeu d'instructions du watchdog

Annexe -B-: Développement des outils pour la génération du programme du watchdog

Annexe -C-: Evaluation de la criticité en fonction des options d'optimisation

Introduction générale

De nos jours, un nombre croissant d'applications reposent sur l'utilisation de systèmes embarqués qui comportent à la fois du matériel et du logiciel. Dans la suite de ce manuscrit, nous nous intéresserons à des systèmes embarqués composés principalement d'un microprocesseur mono-cœur exécutant un logiciel d'application (potentiellement avec le support d'un système d'exploitation). Les co-processeurs spécialisés et les autres composants d'entrée-sortie du système ne seront pas explicitement considérés.

Ces systèmes sont soumis le plus souvent à de fortes contraintes de sûreté de fonctionnement car ils peuvent mettre en jeu des intérêts financiers, industriels et même des vies humaines. La sûreté de fonctionnement est définie comme «la propriété permettant aux utilisateurs de placer une confiance justifiée dans le service délivré » **[Lapr. 85].** Elle dépend directement de la capacité d'un système à réagir de façon sûre (c'est à dire, non dangereuse pour l'application) lorsque des fautes surviennent en cours d'exécution, et avant qu'elles ne provoquent des dysfonctionnements inacceptables du point de vue de l'utilisateur.

Les systèmes critiques ne sont pas en effet à l'abri d'interférences naturelles ou malveillantes qui peuvent provoquer des fautes transitoires et en conséquence perturber leur comportement. Même si les fautes transitoires sont par nature éphémères, des résultats erronés, dont la gravité peut être très grande, peuvent avoir été générés pendant l'intervalle de temps de la perturbation et se propager ensuite dans le système. Les causes d'une telle faute sont diverses (radiations, couplage entre lignes de transmission proches, attaques volontaires, …) et ses conséquences peuvent être classées principalement en deux catégories :

- Erreurs sur les données : ce type d'erreur apparait quand la valeur d'une variable, en mémoire ou stockée dans un registre, est altérée.

- Erreurs sur le flot de contrôle : ce type d'erreur apparait quand le contenu d'une case mémoire ou d'un registre stockant une instruction ou un état est altéré. Pour le type de système considéré, les erreurs de flot de contrôle se manifestent strictement parlant par l'exécution d'une séquence d'instructions incorrecte. Par extension, ceci inclut les erreurs dans les codes des instructions qui sont lues (par exemple, si une addition est transformée en une soustraction, même si le séquencement des instructions reste correct, cette erreur est considérée comme une erreur de flot de contrôle). D'après une étude de Schmid et al. **[Schm. 82]** en 1982, les erreurs de flot de contrôle qui surviennent pendant le fonctionnement d'une application peuvent atteindre 80% du total des erreurs pour certains types de systèmes.

Dans ce document, nous nous intéresserons aux deux types d'erreurs et à des protections qui peuvent être implantées pour limiter leurs effets. Pour le second type d'erreur, les protections sont dîtes "méthodes de vérification des flots de contrôle" (ou Control Flow Checking - CFC). Ce type de vérification sera ici considéré au niveau système, afin de vérifier que les instructions du

programme d'application du microprocesseur sont lues sans erreur et dans le bon ordre. Les erreurs sur les données seront prises en compte par une extension de la vérification de flot de contrôle. L'objectif sera uniquement de détecter un mauvais comportement du système ; la correction de l'exécution, ou la ré-initialisation du système, pourra alors être activée par d'autres mécanismes. En revanche, un mauvais comportement devra pouvoir être détecté avec une grande probabilité sans faire d'hypothèses simples sur les types d'erreur pouvant survenir, et notamment sur le nombre de bits pouvant être perturbés.

Toutefois, le surcoût lié à ces protections doit être gardé dans des limites acceptables sur le plan économique. Pour cela, un partitionnement entre logiciel et matériel et une optimisation de l'interface entre les fonctions réalisées aux deux niveaux, sont à envisager pour allier efficacité et faible coût. Il est également important de prendre en compte les contraintes d'implantation typiquement rencontrées dans les systèmes embarqués, notamment la nécessité de séparer physiquement les fonctions applicatives et les fonctions de surveillance, ainsi que le maintien des performances et de l'intégrité du système à surveiller. Ceci implique d'une part que le système initial ne doit pas être modifié ; la surveillance doit être effectuée par des dispositifs positionnés en parallèle du système et connectés uniquement pour effectuer une observation du système. D'autre part, les dispositifs de surveillance doivent pouvoir fonctionner au rythme de l'application et ne pas entraîner de ralentissements notables, qui pourraient induire des violations des contraintes temps réel.

Le chapitre 1 de ce manuscrit sera consacré, dans sa première partie, à une présentation des notions générales de la sûreté de fonctionnement. Dans un premier temps, nous étudierons les origines, les conséquences et les modèles des dysfonctionnements qui peuvent survenir dans un circuit. Par la suite, nous allons parcourir l'éventail des solutions de détection de ces dysfonctionnements. La deuxième partie de ce chapitre sera dédiée quant à elle à l'état de l'art des techniques de vérification de flot de contrôle. Celui-ci nous permettra de faire une synthèse des avantages et des inconvénients des techniques existantes et d'affiner nos choix par rapport à la solution à concevoir.

A partir des connaissances acquises avec l'étude de l'état de l'art, une nouvelle méthode de vérification de flot de contrôle est proposée, nommée IDSM. Le chapitre 2 portera sur la présentation de cette méthode, ainsi que sur les différentes solutions suggérées pour que cette méthode puisse être implantée avec un microprocesseur moderne, pouvant posséder des mécanismes gênants pour la vérification du flot de contrôle et exécutant un programme complexe avec des appels à des fonctions externes.

Le chapitre 3 aura pour but de présenter un prototype exploitant la méthode de vérification de flot de contrôle proposée. Pour ce faire, nous décrirons l'architecture adoptée ainsi que les outils développés. Nous évaluerons ensuite l'efficacité de cette méthode ainsi que ses différents surcoûts. Ces évaluations mettront également en lumière les avantages les plus marquants d'IDSM.

Dans le chapitre 4, nous aborderons la problématique de l'augmentation de la robustesse sous un angle complémentaire. En effet, de nombreuses techniques, dont IDSM, ont été proposées pour réduire la vulnérabilité des systèmes par la détection des fautes en ligne, mais rares sont les études qui se sont occupées de l'amélioration intrinsèque de la robustesse pendant la génération de l'application logicielle. Disposant des critères de criticité définis dans le chapitre 2, et des outils nécessaires pour les évaluer, présentés dans le chapitre 3, nous analyserons les effets des options de compilation sur la criticité des variables. L'objectif est d'identifier les options qui réduisent la criticité, ou au contraire nuisent à la criticité globale, que les variables soient stockées en mémoire ou dans les registres.

Chapitre I :

Etat de l'art

Les systèmes embarqués sont le plus souvent basés sur des microprocesseurs ou microcontrôleurs exécutant un logiciel d'application et sensibles à de nombreuses perturbations, tant au niveau atmosphérique que dans l'espace. Ils sont souvent utilisés pour des applications critiques, de plus en plus exigeantes en termes de sûreté de fonctionnement.
Dans ce chapitre, nous aborderons les sources, les conséquences et les modèles des dysfonctionnements qui peuvent survenir en opération dans ces systèmes embarqués (I.1 et I.2). Par la suite, et après une vue d'ensemble sur les techniques de détection des erreurs (I.3), nous étudierons quelques méthodes présentes dans la littérature qui portent sur la vérification de flot de contrôle (I.4), la vérification des données (I.5) ou encore les méthodes combinées flot de contrôle et données (I.6).

I.1. Origines et conséquences des fautes transitoires et des dysfonctionnements temporaires (dont attaques)

I.1.1. Les origines

Le mauvais fonctionnement d'un circuit peut résulter de problèmes liés soit à sa conception et fabrication, par exemple quand celles-ci ne sont pas conformes à la spécification, soit à une perturbation à cause de l'environnement (interférences, bruits, radiations, particules…) ou de son vieillissement (dégradation des caractéristiques). Dans ces travaux nous n'allons considérer que la deuxième source de mauvais fonctionnement, qui est relative aux problèmes de perturbation des applications en ligne.

Parmi les problèmes d'exécution liés à l'environnement, on distingue d'une part les perturbations d'origine naturelle, qui font l'objet d'études dans le domaine du test (particulièrement du test « en-ligne »), et d'autre part celles dont les origines sont intentionnelles, liées au domaine de la sécurité. Bien que ces deux domaines aient des espaces d'application très différents, la méthode de durcissement proposée doit pouvoir faire face aux fautes quelques soient leurs origines naturelles ou intentionnelles.

I.1.1.1. Environnement du système :

Origines naturelles :

Les applications spatiales et aéronautiques sont très sensibles aux rayonnements cosmiques car elles évoluent dans des environnements sensibles. Mais l'évolution des technologies dans le domaine "submicronique profond" s'accompagne d'un fort accroissement de la sensibilité des circuits à différents parasites (rayonnements ou particules), même au niveau de la mer. Par conséquent, le problème de la sensibilité aux rayonnements concerne aujourd'hui les

applications grand public. Les constructeurs et équipementiers automobiles, comme les fournisseurs de routeurs internet, sont par exemple fortement concernés par cette problématique.

A part ces rayonnements cosmiques, il y a dans l'environnement d'un circuit d'autres sources de radiations qui peuvent produire des fautes dont notamment les impuretés radioactives dans le boîtier du circuit, ou les interactions Neutron-Boron10 **[Baum. 01]**.

Origines intentionnelles :

Outre les fautes d'origine naturelle il faut désormais prendre en considération les fautes d'origine intentionnelle. En effet, alors que la sécurité est une propriété de plus en plus recherchée pour de nombreuses applications (cartes de crédit, télécommunications, voire électronique grand public …), il devient indispensable de considérer les attaques intentionnelles utilisées pour modifier le comportement d'un système et ainsi obtenir des droits supplémentaires ou des données censées rester secrètes comme une clé de chiffrement **[BarE. 06]**.

Des fautes peuvent être injectées en modifiant l'environnement du circuit et ce de différentes manières : variation de la tension d'alimentation ou de la température, variation de l'horloge ou du reset, ou encore injections optiques (flash, laser).

I.1.1.2. Perte de l'intégrité du signal

L'intégrité du signal est la qualité des signaux engendrés dans un circuit et leur capacité à bien représenter les valeurs logiques souhaitées. L'intégrité d'un signal peut éventuellement être altérée par des interférences dues aux autres éléments du circuit ou du système. Cette notion peut être étendue pour prendre en compte tout type d'interférence y compris celles provenant de rayonnements électromagnétiques, voire de rayonnements cosmiques **[Angh. 00]**.

Les problèmes d'intégrité du signal (couplages, bruits …) sont de plus en plus pris en compte lorsque l'on étudie les phénomènes de fautes des systèmes. Plus prévisibles que les impacts de particules, ils augmentent avec la réduction des dimensions. Parmi les causes de perte de l'intégrité du signal il y a notamment la diaphonie capacitive et inductive **[Angh. 00]**, la pollution des alimentations et les variations de l'horloge **[Vanh. 08]**.

I.1.2. Les conséquences

Une faute peut rester latente dans le cas où elle se trouve dans une partie non utilisée du circuit, ou si elle est temporairement masquée lors du calcul en cours. Dans le cas contraire elle est activée et engendre une erreur. Cette dernière peut être définie comme étant un état anormal du système et peut entraîner à son tour une défaillance si elle se propage et devient observable de l'extérieur **[Lapr. 04]**, la défaillance étant une déviation du service fourni par rapport au service correct **[Cour. 91]**. Cet enchaînement est illustré dans la Figure 1-1.

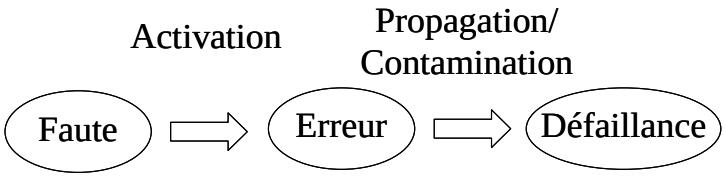


Figure 1-1 : Schéma Faute-Erreur-Défaillance

On distingue trois grandes catégories d'erreurs : les erreurs transitoires, intermittentes et permanentes.

- Une erreur transitoire (*soft-error*) est caractérisée par une modification réversible de donnée ou un état erroné temporaire. Elle n'engendre pas de destruction et le circuit fonctionne à nouveau normalement après la disparition de cette erreur. Par exemple, une erreur transitoire dans un registre disparaît après réécriture de ce registre. Mais elle peut s'être propagée entre temps en créant d'autres erreurs dans le système.
- Une erreur intermittente survient de manière répétée mais non systématique. Une telle erreur est souvent liée à un fonctionnement dans des conditions particulières (par exemple, à une certaine température, pour certaines valeurs des données traitées). Dans la suite, la manifestation d'une erreur intermittente sera assimilée à une erreur transitoire.
- Une erreur permanente (*hard-error*) correspond à la destruction partielle du circuit. Notre étude s'est portée uniquement sur les erreurs transitoires car l'occurrence d'une erreur permanente détectable rendrait le type de système étudié incapable d'assurer l'ensemble de ses missions. De plus, la probabilité d'occurrence d'une erreur transitoire est plus grande.

I.2. Modèles de fautes et d'erreurs

Les modèles de fautes sont une abstraction des perturbations subies par un circuit. Ils sont généralement les intermédiaires entre les phénomènes physiques présentés ci-dessus et les modèles d'erreurs à plus haut niveau nécessaires pour notre étude. Dans de nombreux cas, l'appellation "modèle de fautes" est toutefois employée pour indiquer des modèles d'erreurs.

Le plus souvent, un impact de particule est très localisé. On a donc un effet singulier ou SEE (Single Event Effect), qui peut être de deux types principaux :

- SEU (Single Event Upset) : inversion d'un bit dans la logique séquentielle ou une mémoire.
- SET (Single Event Transient) : inversion temporaire d'un signal dans les parties analogiques ou la logique combinatoire.

Notons que dans le premier cas il s'agit de la création directe d'une erreur (modification de l'état du circuit) alors que dans le second cas il s'agit d'une faute, qui doit être sensibilisée pour éventuellement générer une erreur.

La probabilité que l'impact d'une particule touche plusieurs nœuds d'un même circuit était auparavant négligeable mais doit aujourd'hui être prise en compte. En effet, une particule peut, par exemple, influencer dans sa course plusieurs transistors associés à plusieurs cellules mémoire (phénomène de multi-collection). Dans ce cas, on observe des effets multiples tels que les MBU et les MCU (Multiple Bit Upset - MBU ou *Multiple Cell Upset* – MCU). Les MBU sont généralement définis comme plusieurs bits erronés dans un même mot mémoire ou un même registre, tandis que les MCU correspondent à plusieurs bits erronés dans des mots mémoire différents. La propagation d'un SET (ou de SET multiples liés à la multi-collection) peut aussi conduire à inverser des bits multiples, dans un ou plusieurs registres.

Nous considérerons donc dans la suite un modèle général nommé MBF (Multiple Bit Flip), qui englobe ces derniers modèles ainsi que les SEU, comme étant un cas particulier avec une multiplicité égale à 1. Dans le modèle MBF, la multiplicité peut être quelconque ainsi que la position des bits erronés.

I.3. Vue d'ensemble sur les techniques de détection

La détection de fautes consiste à déceler et à signaler l'occurrence d'une faute, susceptible d'altérer le service fourni par le circuit. Dans de nombreux cas, la détection est davantage une détection d'erreur, mais comme indiqué précédemment le terme "faute" est souvent utilisé de manière générique pour dire "faute" ou "erreur". Certaines techniques de durcissement permettent de rendre un circuit résistant aux fautes ; l'évitement de fautes est le plus souvent lié à des étapes technologiques. Ce qui nous intéresse dans ce manuscrit concerne les techniques appliquées pendant la conception logique du système. Les fautes sont alors détectées par des mécanismes ajoutés au circuit et celui-ci en informe son environnement.

I.3.1. Redondance matérielle

La redondance matérielle consiste à répliquer le circuit à protéger, pour que les calculs soient effectués deux fois (ou davantage). S'il y a une inconsistance des résultats obtenus cela signifie qu'une erreur est survenue. C'est la technique de détection la plus simple. Le coût en surface et en consommation est au-delà de +100% pour une simple duplication. Le coût en vitesse est faible car le signal de comparaison est obtenu avec de la logique très simple et en parallèle du signal de sortie. Cette architecture peut s'appliquer à tous les blocs mais compte tenu de ses coûts elle est l'une des moins optimisées.

Plusieurs variantes de redondance existent telle que la Duplication Avec Redondance Complémentaire (DWCR : Duplication With Complementary Redundancy). Cette technique est similaire à la méthode classique et présente sensiblement les mêmes coûts. La différence est que les signaux et les données internes dans les deux modules dupliqués doivent être de polarités opposées. Néanmoins cette méthode augmente la complexité de la conception par rapport à une duplication simple. La réalisation en double rail (DRC : Double Rail Code), dans laquelle les deux sorties sont inversées s'il n'y a pas de fautes, peut être vue comme un autre cas de redondance complémentaire.

I.3.2. Redondance temporelle

Dans le cas d'une redondance temporelle, l'opération est effectuée deux fois de suite et les résultats sont comparés. Cette méthode est simple, et bien adaptée aux fautes transitoires comme les SEU. Cette architecture entraîne un surcoût en performance très important car le temps de calcul est plus que doublé. Le surcoût en surface est limité à la mémorisation du premier résultat et à la logique de comparaison, la puissance consommée augmente peu (mais l'énergie augmente notablement).

I.3.3. Redondance d'information

La redondance d'information consiste à répliquer une information ou lui ajouter des métadonnées pour détecter des erreurs qui peuvent la toucher. Par exemple, une donnée critique peut être reproduite en mémoire ou dans une mémoire auxiliaire pour que sa valeur puisse être vérifiée avant chaque utilisation.
Parmi les métadonnées qui peuvent être utilisées nous pouvons citer les codes détecteurs d'erreurs dont notamment le codage par parité, les codes non ordonnés ou le codage résidu.

I.3.4. Contrôles temporels et d'exécution

Des techniques particulières de protection peuvent être définies pour certains blocs. Parmi ces techniques il existe la vérification du flot de contrôle par analyse de signature (CFC : Control Flow Checking). Un CFC permet de vérifier que les instructions d'un programme sont lues sans erreur et dans le bon ordre. Appliqué au niveau matériel, un CFC peut aussi permettre de vérifier l'absence de sauts non prévus entre deux états d'une machine à états finis. Au niveau d'un système d'exploitation, le CFC peut permettre de détecter des changements illégaux dans l'ordre d'exécution des tâches. De nombreuses techniques existent et peuvent être implantées soit uniquement au niveau logiciel, soit uniquement au niveau matériel, soit en faisant collaborer des modifications du logiciel et un élément de support matériel (coprocesseur de vérification "watchdog" ou "IP d'infrastructure"). Les méthodes CFC présentent le meilleur compromis entre le taux de couverture des erreurs et les différents surcoûts, notamment lorsque la multiplicité des erreurs n'est pas fixée à priori. C'est pour cette raison que la méthode de durcissement que nous proposerons dans cette thèse s'appuiera sur une méthode de vérification de flot de contrôle. Nous allons donc plus particulièrement détailler cette partie de l'état de l'art.

I.4. Vérification de flot de contrôle

Avant d'entrer dans le vif du sujet, il est important de définir quelques notions théoriques sur les méthodes de vérification de flot de contrôle, notamment les termes techniques, afin de fournir au lecteur toutes les bases nécessaires pour appréhender le travail qui sera détaillé dans les chapitres suivants. Pour illustrer toutes ces notions, nous considèrerons le programme qui calcule le plus grand commun diviseur PGCD de deux entiers x et y (figure 1-2-A).

I.4.1. Principes généraux des méthodes de vérification de flot de contrôle

I.4.1.1. Graphes de flot de contrôle (GFC)

Tout flot de contrôle peut être représenté par un graphe (GFC) orienté et noté G={N, A} (tel que N est l'ensemble des nœuds de ce graphe et A l'ensemble des arcs les reliant) et cela indépendamment du niveau d'abstraction étudié, circuit ou système.

- Au niveau circuit, le GFC a la même structure que l'automate définissant le contrôleur :
  - Chaque nœud du GFC correspond à un état de l'automate, et son identificateur n'est autre que le code de cet état.
  - Les arcs du GFC sont les arcs présents dans le graphe de cet automate (les prédicats restent identiques).
- Au niveau système, le GFC peut être rapproché de l'organigramme du programme d'application.
  - Chaque instruction du programme est représentée par un nœud du GFC, à qui sont associés le numéro et le code de cette instruction (voir figure 1-2-B)
  - Les arcs du GFC représentent le passage d'une instruction à une autre, et les prédicats qui leur sont associés correspondent aux conditions de branchement dans l'organigramme (exemples de la figure 1-2-B : x<y, x>y ou x=y)

  Contrairement à ce que l'on trouve dans les automates et les GFC relatifs au niveau matériel, aucune opération ne peut être effectuée sur les arcs. Par contre, une nouvelle notion apparaît dans les GFC du niveau système, celle de la hiérarchisation du GFC. Cette hiérarchisation est introduite lorsque le programme d'application représenté effectue des appels aux sous programmes.

Outre la représentation du GFC illustrée dans la figure 1-2 et décrite plus haut (où chaque instruction correspond à un nœud du GFC), il existe une deuxième représentation, illustrée dans la figure 1-3 et correspondant à une décomposition en blocs linéaires, divergences et convergences vers des points de jonction.

Définition 1 : Bloc linéaire : c'est un sous graphe orienté possédant les caractéristiques suivantes :

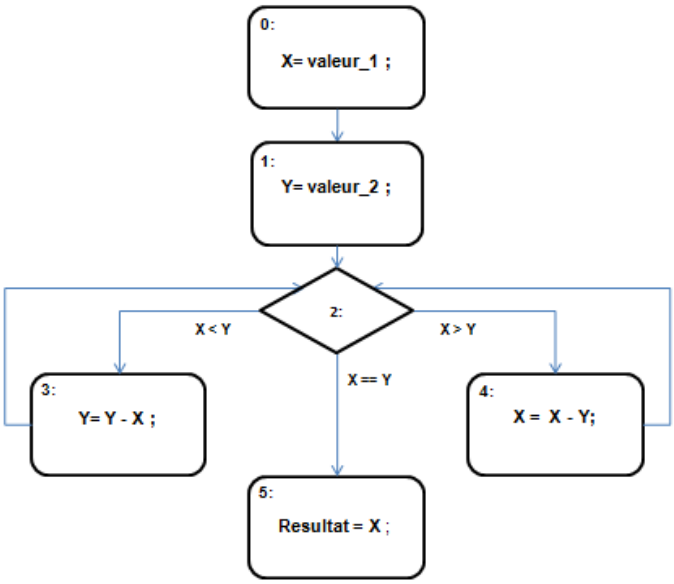
- Le premier nœud du bloc est le seul point d'entrée,
- Le dernier nœud du bloc est l'unique point de sortie,
- Aucune divergence n'existe à l'intérieur du bloc,
- Si l'opération représentée par le premier nœud du bloc est exécutée, alors toutes les opérations associées aux divers nœuds du bloc sont exécutées dans l'ordre indiqué par l'orientation des arcs.

Définition 2 : Divergence : C'est un ensemble d'arcs qui ont la même origine mais des destinations différentes. Elle correspond à un branchement conditionnel.
Les prédicats associés aux arcs d'une divergence sont supposés tous différents et non intersectants et doivent généralement former une tautologie. Exemple de la figure 1-3 :

$$(x<y) \cap (x>y) \cup (x<y) \cap (x==y) \cup (x>y) \cap (x==y) = \varnothing$$

Définition 3 : Point de jonction : c'est la destination d'un ensemble d'arcs d'origines différentes.

```
0:  x = valeur_1 ;
1:  y = valeur_2;
2:  Tant que  (x != y) faire {
    si (x < y)
3:          y = y - x;
    sinon
4:          x = x - y;
    }
5:  Résultat = x ;
```



-A- -B-

Figure 1-2 : Exemple d'application modélisée en GFC

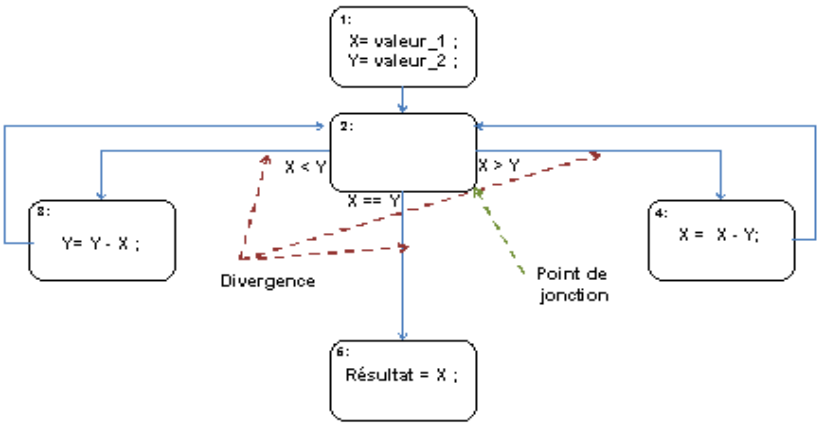


Figure 1-3 : Structure d'un GFC

Les méthodes de vérification de flot de contrôle consistent en une comparaison entre un GFC de référence et le flot de contrôle obtenu lors du fonctionnement du système à surveiller.
Pour qu'une exécution soit considérée comme correcte, deux conditions doivent être vérifiées :

1. Le chemin parcouru lors de l'exécution est **légal** et **correct** par rapport au GFC de référence.
2. Les opérations spécifiées sur chaque nœud sont exécutées correctement.

Définition 4 : Chemin légal : Un chemin est considéré légal si toutes ses transitions suivent des arcs existants dans le GFC (Voir figure 1-4).

Définition 5 : Chemin correct : Un chemin est considéré correct s'il est légal et si toutes ses transitions sont autorisées, autrement dit vérifiant le prédicat associé à l'arc (Voir figure 1-4).

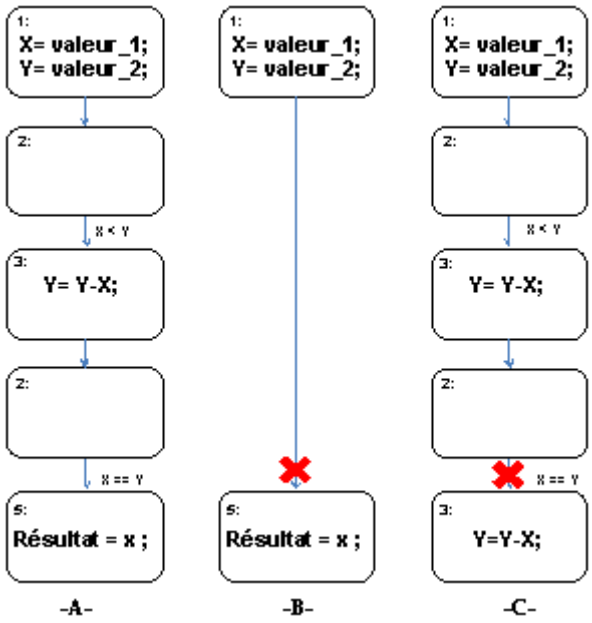


Figure 1-4 : Exemple de chemin dans un GFC (4-A chemin correct, 4-B chemin illégal, 4-C chemin incorrect)

En règle générale, il n'est pas possible de s'assurer que l'exécution est effectivement correcte. La surveillance est donc limitée au mieux à l'identification des chemins incorrects ou illégaux et à la bonne correspondance des opérations demandées sur chaque nœud du chemin exécuté.
Pour résumer, la vérification du flot de contrôle nécessite :

- La mémorisation du GFC de référence du système,
- L'extraction du flot de contrôle du système en cours de fonctionnement,
- La comparaison entre le GFC de référence et le flot de contrôle pendant le fonctionnement du système à surveiller. La fréquence de ces comparaisons détermine le temps de latence minimal de détection d'une erreur par le dispositif de surveillance.

I.4.1.2. La compaction d'information

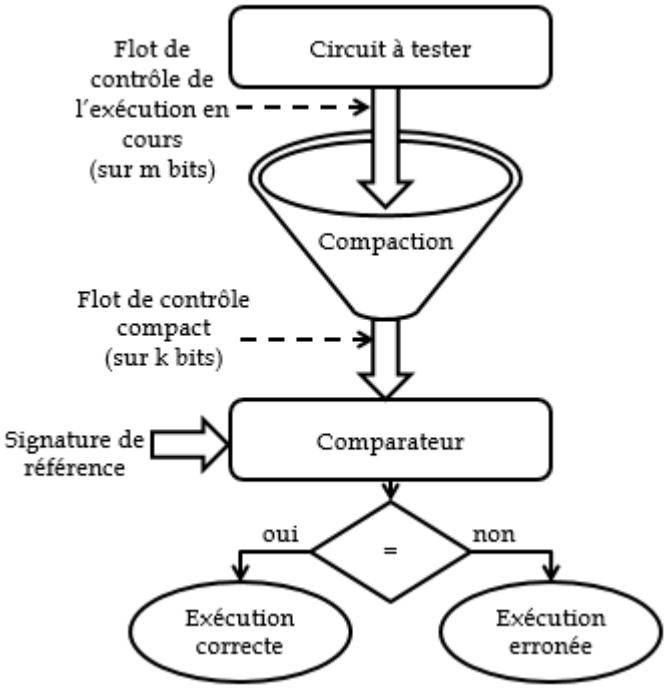


Figure 1-5 : Principe du test en ligne avec compaction d'information

La quantité d'informations nécessaire pour effectuer les tests en ligne est souvent prohibitive. Il est donc indispensable d'utiliser une représentation sous une forme réduite désignée par le nom de « signature ». Cette réduction est appelée « compaction ».

L'objectif est essentiellement de réduire la quantité d'informations à stocker pour pouvoir effectuer, en ligne, la comparaison avec le GFC de référence. Bien évidemment, le circuit compacteur, ayant généré les signatures de référence, doit également être placé en aval du circuit surveillé afin de calculer en ligne les signatures correspondant aux signaux à surveiller. (Voir figure 1-5).

Il existe 3 types de compacteurs :

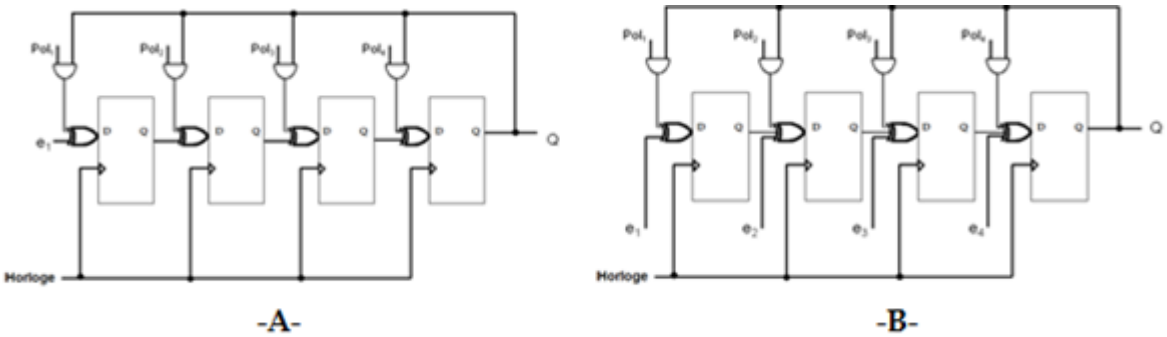
1. Compacteur spatial : il permet à chaque cycle d'avoir une signature correspondant à une fonction des différents signaux observés. Dans ce cas $k < m$.
2. Compacteur temporel : il permet d'obtenir pour chaque signal une signature de la séquence de valeurs obtenue pendant un certain nombre de cycles nb_c. Dans ce cas $k_i < nb_c$ pour chaque signal i.
3. Compacteur spatio-temporel : il calcule une signature à la fois spatiale et-temporelle. Dans ce cas $k < nb_c * m$.

Définition 6 : Signature verticale : c'est le résultat de la compaction spatio-temporelle d'un bloc du GFC, autrement dit c'est la forme réduite de toutes les instructions d'un seul bloc.

Définition 7 : Signature horizontale : c'est le résultat de la compaction spatiale d'une seule instruction.

Le choix d'une fonction de compaction pour un dispositif d'analyse de signature est important car il détermine le taux de masquage, c'est-à-dire le taux d'erreurs indétectables, à cause de la perte d'information entraînée par la compaction. Par ailleurs, dans les techniques de test intégré, le dispositif de compaction se doit d'être particulièrement simple pour permettre un coût matériel aussi faible que possible. Il existe plusieurs fonctions de compaction, mais nous résumons ici brièvement la méthode la plus employée, qui est basée sur une division

polynomiale (une présentation plus détaillée et plus complète se trouve dans **[Leve. 90]**). Pour une compaction par division polynomiale, le coût matériel dépend de la structure du dispositif et de la manière de réaliser le polynôme diviseur.



-A- -B-

Figure 1-6 : Exemple de circuit de compaction par division polynomiale (6-A LFSR, 6-B MISR)

Pour la compaction d'un flot d'information parallèle, le dispositif (figure 1-6-B) est appelé MISR (Multiple Input Shift Register). L'appellation de LFSR (Linear Feedback Shift Register) est réservée aux circuits de compaction par division polynomiale d'un flot d'information série (figure 1-6-A). Les coefficients du polynôme, indiqués sur la figure, peuvent être programmables ou fixes ; dans ce dernier cas la structure est simplifiée.
La probabilité de masquage avec cette technique et un polynôme diviseur premier primitif tend vers 2^{-k}, k étant le nombre de bits de la signature, lorsque la séquence à analyser est longue (nombre de bits compactés tendant vers l'infini, ou du moins suffisamment grand).

I.4.1.3. Traitement des exceptions

Comme nous l'avons vu dans le paragraphe I.4.1.1, la vérification du flot de contrôle n'est autre que la comparaison entre un GFC de référence et le flot de contrôle pendant le fonctionnement du système à surveiller. Mais ce qui n'a pas été dit c'est que le GFC de référence ne peut pas nous renseigner sur les exceptions qui peuvent survenir pendant le fonctionnement du système. De ce fait les exceptions doivent être traitées à part pour assurer l'invariance de la signature au départ et au retour d'une exception. Il faut donc considérer les 3 problèmes suivants **[Mich. 93]** :

- La reconnaissance d'un départ en exception : La reconnaissance d'un départ en exception peut en général se faire grâce aux informations placées par le processeur sur ses signaux externes au moment du départ en exception (demande de cycle de vectorisation, accès à une table de vecteurs), mais ceci est différent pour chaque processeur. Par ailleurs, certains processeurs ne signalent pas toutes leurs exceptions logicielles en particulier lorsque celles-ci sont de type « auto-vectorisé », en général sur défaut logiciel pendant le traitement d'une instruction (division par zéro …). De plus pour certains processeurs, l'adresse de la table des vecteurs d'exception n'est pas fixe donc la détection d'un accès à cette table n'est pas réellement envisageable pour la reconnaissance des départs en exceptions.
- Les traitements associés au départ en exception : Une fois l'exception reconnue, le dispositif de vérification de flot de contrôle doit empiler la signature courante, pour être en mesure de reprendre le calcul de la signature lors du retour de l'exception.

- Les traitements associés au retour de l'exception : Les traitements effectués par le dispositif de vérification au niveau d'un retour d'exception sont assez simples en comparaison du cas d'un départ. La reconnaissance ne pose aucun problème et le rôle du dispositif de vérification est donc simplement de comparer la signature de la routine d'exception puis de recharger le dispositif de compaction avec la signature empilée au départ.

I.4.1.4. Modélisation fonctionnelle des erreurs de flot de contrôle

D'après la définition dans **[Bens. 03]**, toute altération du contenu d'une case mémoire ou d'un registre stockant une instruction est considérée comme étant une erreur de flot de contrôle.
A partir de cette définition nous pouvons distinguer deux types d'erreurs :

- Erreurs de séquencement : Ce sont les erreurs qui modifient le séquencement des instructions du programme.
- Erreurs sur le contenu de l'instruction : ces erreurs modifient le sens d'une instruction par corruption des codes opératoires ou des opérandes.

Les erreurs de séquencement peuvent être classées en deux catégories :

1) Erreurs sur les instructions de branchement **[Bori. 06] :**
   - A. Chemin incorrect
   - B. Saut vers le début du même bloc
   - C. Saut vers le milieu du même bloc
   - D. Saut illégal vers le début d'un autre bloc
   - E. Saut vers le milieu d'un autre bloc
   - F. Saut vers une zone mémoire non allouée au programme.

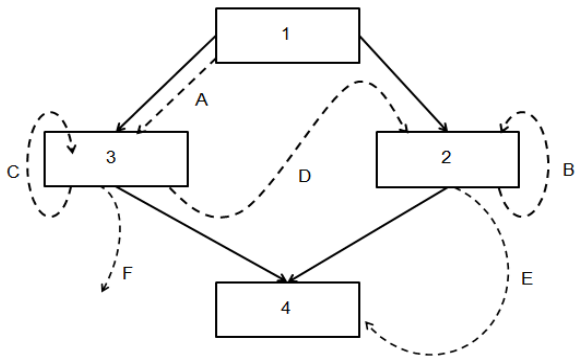


Figure 1-7 : Erreurs sur les instructions de branchement

2) Erreurs sur les instructions de non branchement
   - A'. Saut vers le milieu d'un autre bloc
   - B'. Saut vers le début du même bloc
   - C'. Saut vers le milieu du même bloc
   - D'. Saut vers le début d'un autre bloc
   - E'. Saut vers une zone mémoire non allouée au programme.

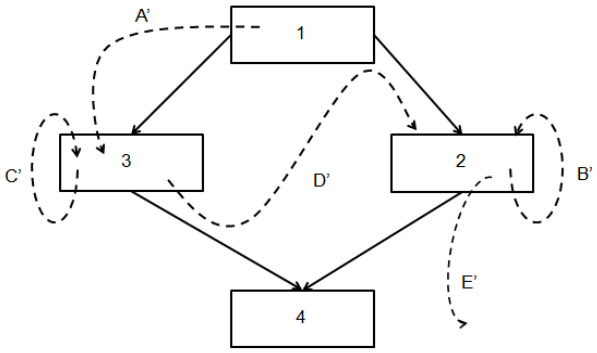


Figure 1-8 : Erreurs sur les instructions de non branchement

Ces erreurs peuvent survenir pendant l'exécution du programme mais aussi pendant les appels des sous programmes et des routines d'interruption.

I.4.2. Classification des méthodes de vérification de flot de contrôle

Les méthodes de vérification de flot de contrôle peuvent être classifiées en deux catégories selon la manière avec laquelle les données relatives aux signatures et les références de comparaison sont stockées **[Mich. 94] :**

- Les méthodes ESM (Embedded Signature Monitoring) : les données relatives aux signatures sont directement enfouies dans le programme principal en tant que paramètres ou instructions spécifiques. Ces méthodes sont les plus utilisées car l'insertion des informations directement dans le programme d'application permet de s'assurer très simplement des instants où les signatures doivent être calculées et vérifiées. Elles ont toutefois plusieurs inconvénients :
  - La modification systématique du programme vérifié peut être une source d'erreur en elle-même, elle tend à diminuer la portabilité de l'application, et elle nécessite une nouvelle vérification du système après insertion des instructions de surveillance, ce qui peut être très coûteux.
  - Le processeur est ralenti à cause des NOP ou autres opérations qu'il doit faire lors de la rencontre d'une instruction spécifique ; au-delà de la perte de performances, des contraintes temps réel peuvent alors être violées alors qu'elles ne l'étaient pas dans le système initial.
- Les méthodes DSM (Disjoint Signature Monitoring) ou ASM (Autonomous Signature Monitoring) : dans ce cas les informations concernant la surveillance du flot de contrôle doivent être disjointes et stockées dans une autre partie de la mémoire (ou plus souvent une autre mémoire) avec suffisamment d'informations sur la structure du programme d'application pour pouvoir identifier à la volée les instants où des compactions sont à réaliser et ceux où des signatures doivent être comparées. Un squelette du programme d'application doit donc être stocké en plus des données relatives aux signatures, ce qui entraîne un grand surcoût en mémoire. Le bloc matériel chargé de l'analyse de signature doit aussi être plus complexe. Toutes ces raisons ont fait que les recherches sur les méthodes DSM n'ont pas été très nombreuses, bien que cette approche permette de supprimer les inconvénients cités pour les approches ESM.

I.4.2.1. Méthodes de vérification de flot de contrôle avec signatures enfouies

I.4.2.1.1. Méthodes de vérification de flot de contrôle purement logicielles

Les méthodes de vérification de flot de contrôle purement logicielles consistent à modifier le programme à vérifier en lui ajoutant des instructions de calcul et de comparaison de signatures en ligne. Ces ajouts peuvent être effectués pendant la compilation ou pendant une phase de pré-processing. Parmi les méthodes récentes, nous pouvons notamment citer CFCSS **[Oh 02]**, ECCA **[Alkh. 99]** et CEDA **[Vemu 06]**. Dans ces trois méthodes, les insertions d'instructions sont réalisées automatiquement grâce à la modification du compilateur, en l'occurrence GCC pour

les méthodes ECCA et CEDA. Ceci offre à ces méthodes plus de portabilité et moins de surcoût par rapport aux autres solutions utilisant un préprocesseur.

Control Flow Checking by Software Signatures CFCSS :

La CFCSS **[Oh 02]**, utilise un registre GSR (Global Signature Register) dédié au stockage de la signature courante G_i associée au bloc en cours d'exécution. Une signature unique S_i est associée à chaque bloc V_i. Si l'exécution est correcte, S_i est égale à G_i. Quand le contrôle est transféré d'un bloc à un autre, une nouvelle signature globale G est calculée en fonction de la signature du bloc courant V_i, des signatures des nœuds V_j précédents tel que $V_j \in prec(V_i)$ et d'un paramètre d_i spécifique à chaque bloc et défini pendant la compilation.

$$G_i = G_j \oplus d_i = G_j \oplus (S_i \oplus S_j)$$

Avec ce calcul, quand on veut accéder à V_2 à partir de deux autres blocs V_3 ou V_4 une erreur sera en général déclenchée. Il faut donc ajouter un ajustement avant le déclenchement de l'erreur.

Pour cela, la technique CFCSS utilise une variable « D », mise à jour dans chaque bloc dont la destination possède plusieurs prédécesseurs.

- Pour le nœud V_{j1} choisi arbitrairement ➔ D=0000
- Pour tous les autres prédécesseurs

$V_{jm} \neq V_{j1}$ ➔ $D= S_{j1} \oplus S_{jm}$

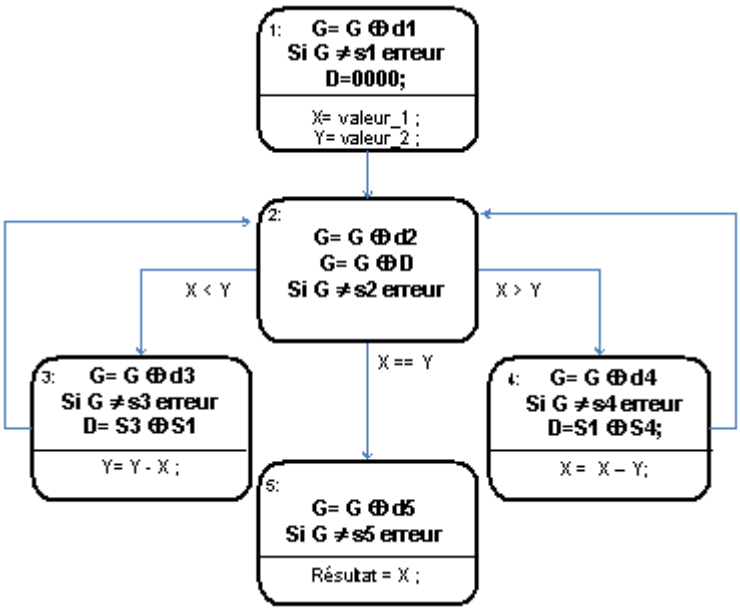


Figure 1-9 : Vérification du flot de contrôle avec CFCSS

La méthode CFCSS est capable de détecter la majorité des branchements illégaux inter-blocs sauf les branchements venant du milieu d'un bloc vers le début du bloc suivant

Il existe une autre erreur de flot de contrôle qui peut survenir sans être détectée : un branchement inter-blocs à partir du milieu ou de la fin d'un bloc Vi vers le milieu du bloc Vj suivant. Pour que cette erreur ne soit pas détectée il faut que le bloc successeur de Vj soit également un successeur de Vi.

Notons qu'une autre approche purement logicielle et affectant une signature unique à chaque bloc avait été proposée vingt ans plus tôt **[Yau 80]**. Ce précurseur avait d'ailleurs proposé l'une des rares méthodes permettant de vérifier la correction des chemins en plus de leur légalité. Mais la méthode avait d'autres limitations.

Enhanced Control-Flow Checking Using Assertions ECCA :

En 1995, Mcfearin et al. ont proposé une méthode de vérification de flot de contrôle, qui s'intitule Control-Flow Checking Using Assertions (CCA) **[Mcfe. 95]**. Cette méthode est purement logicielle et elle appartient à la famille des ESM. Dans CCA, le programme est divisé en plusieurs Branch Free Intervals BFIs auxquels sont assignés deux identifiants :

- Le BID (Branch Free Identifier) : identifiant unique pour chaque BFI.
  - Au début de chaque bloc est insérée l'affectation de cette variable, par exemple BID = 1
  - Et à la fin du bloc est insérée une instruction qui vérifie s'il n' y a pas eu de saut vers un autre bloc avant la fin, par exemple Test BID =1
- Le CFID (Control Flow Identifier) : identifiant identique pour tous les BFIs ayant un parent en commun.

Cette méthode utilise une queue de taille deux et insère 4 instructions par BFI :

- Au début de chaque BFI
  - Affectation du BID
  - Ecriture en queue du CFID du BFI parent.
- A la fin de chaque BFI
  - Lecture en queue du CFID du BFI actuel
  - Test du BID

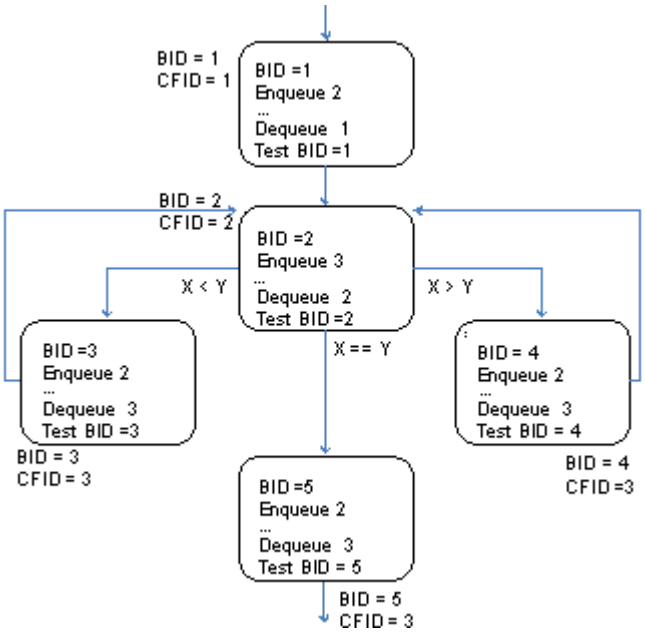


Figure 1-10 : Vérification du flot de contrôle avec CCA

La méthode CCA, bien qu'elle soit très efficace, présente un important surcoût mémoire puisqu'à chaque bloc il faut ajouter quatre instructions. Dans l'optique de diminuer ce surcoût, une nouvelle méthode a été conçue et implémentée, la méthode Enhanced CCA ou ECCA **[Alkh. 99].** Cette technique consiste à ajouter deux instructions « set » et « test » à chaque bloc :

- Set Instruction :

$$Id = \frac{BID}{(!(Id \bmod BID) * (Id \bmod 2))}$$

« Id » une variable globale mise à jour à l'entrée de chaque bloc et « BID » est l'identifiant de chaque bloc et de valeur supérieure à 2

- Test Instruction :

$Id = next + !!(Id - BID)$ «next» est égal au produit des BIDs des blocs accessibles

Dans le cas d'une erreur d'exécution, Id sera différent de BID et donc !!(Id-BID) sera égal à 1. Or comme next est un multiple de BID et ce dernier étant supérieur à 2, Id ne sera plus multiple

de BID. Par conséquent, pendant l'exécution de l'instruction set du bloc suivant, il y aura une division par 0 puisque !(Id mod BID)=0 et donc détection de l'erreur.

Control Error Detection through Assertions (CEDA)

La méthode CEDA **[Vemu 06]** comme toute méthode de vérification de flot de contrôle purement logicielle se base sur l'insertion d'instructions de calcul et de vérification de signatures dans l'application à vérifier.

Dans CEDA, les nœuds sont divisés en 2 catégories A et X : un nœud est de type A si et seulement si il possède plusieurs prédécesseurs et qu'au moins l'un de ses prédécesseurs a plusieurs successeurs sinon le nœud est considéré de type X.

Chaque nœud du GFC est caractérisé par deux paramètres d1 et d2 et identifié par deux signatures NS (Signature Node) et NES (Signature Exit Node) calculées hors ligne.

Un chemin est considéré comme étant correct, si et seulement si, pour chaque nœud i exécuté du GFC deux conditions sont vérifiées :

- S = NSi à tout moment de l'exécution du nœud i
- S = NESi à la sortie de ce nœud

S étant une signature globale calculée en ligne et mise à jour au début et à la fin de chaque nœud grâce aux instructions suivantes :

- Au début du nœud :
  - S= S and d1(N_i)

  si le nœud est de type A
  - S= S xor d1(N_i)

  si le nœud est de type X
- A la fin du nœud :
  - S= S xor d2(N_i)

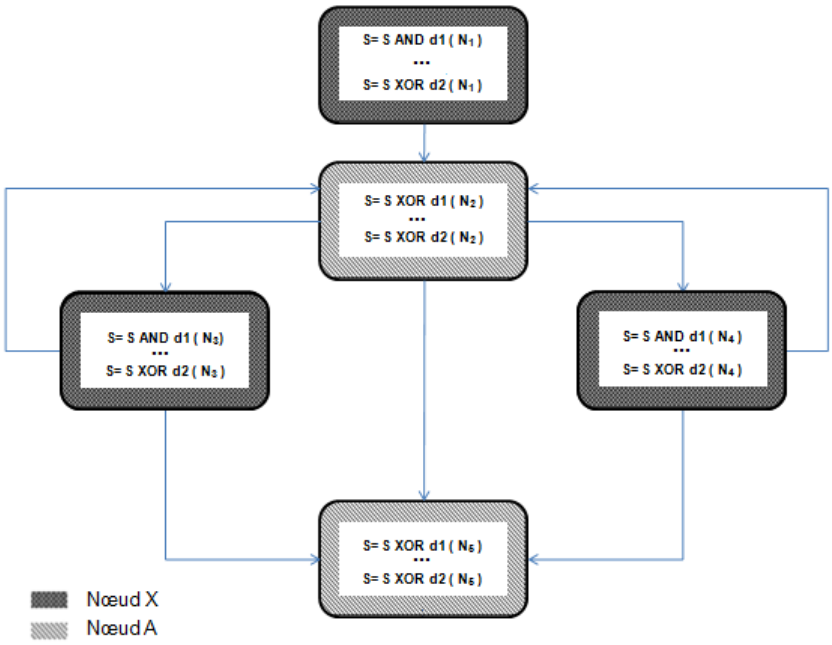


Figure 1-11 : Vérification de flot de contrôle avec CEDA

Une instruction de vérification de la signature « Br S != NES (N_i) error() » est également insérée à divers points du code pour vérifier le flot de contrôle. La fréquence de ces instructions dépendra uniquement du temps de latence maximal désiré.

Deux solutions ont été proposées dans **[Vemu 08]** pour réduire le surcoût en mémoire des méthodes de vérification de flot de contrôle :

- La première solution consiste à créer à partir du GFC des super nœuds ayant une taille choisie selon le taux de couverture désiré.

- La deuxième solution utilise le principe de « localité ». Elle s'inspire de la méthode de conception des mémoires virtuelles des systèmes d'exploitation qui dit que 90% du temps d'exécution d'une application est consacrée à une portion de 10% du code.
  En partant de ce principe, la vérification du flot de contrôle doit être plus importante dans les bouts de code dont l'utilisation est récurrente. Dans le reste de l'application, la vérification sera moins minutieuse, et l'espace mémoire consacré au stockage des données relatives aux signatures sera minimal.

Plusieurs autres approches CFC ont été inspirées de la méthode CEDA, dont notamment les approches de détection et correction des erreurs MCP et CDCC **[Zara. 10]**.
De plus, CEDA a servi comme socle pour la méthode hybride présentée dans **[Azam. 10]** **[Azam. 11]**.

I.4.2.1.2. Détection en ligne des erreurs de flot de contrôle a l'aide d'un IP d'infrastructure [Bern. 05]

Comme nous l'avons déjà vu, les méthodes purement logicielles sont les techniques les plus faciles et les moins coûteuses à implanter au niveau matériel; elles présentent cependant un surcoût significatif en mémoire. Un partitionnement entre logiciel et matériel peut donc être utilisé pour allier efficacité et faible coût. C'est dans cet esprit que la méthode **[Bern. 05]** a été proposée. Cette technique est une solution hybride qui combine l'approche logicielle SIHFT avec l'utilisation d'un IP d'infrastructure baptisé Pandora.
Pandora permet de vérifier le flot de contrôle à l'aide d'une signature B_i associée au bloc V_i en cours d'exécution. Deux types d'instructions sont ajoutés à chaque bloc du programme pendant la compilation :

- Instructions de vérification IIP***Test(Bj)***, ajoutées au début de chaque bloc : pour chaque $V_j \in pred(V_i)$, la signature B_j est envoyée à Pandora, qui se charge de vérifier si Bi est égale à la valeur de référence

- Instruction d'affectation IIPS***et(Bi)***, ajoutée à la fin de chaque bloc: mise à jour de la signature avec la valeur B_i associée à V_i : Bi = (Bj & M1) ⊕ M2,

où « M1 » et « M2 » sont deux constantes calculées pendant la phase de compilation grâce à l'algorithme présenté dans **[Golo. 03].**

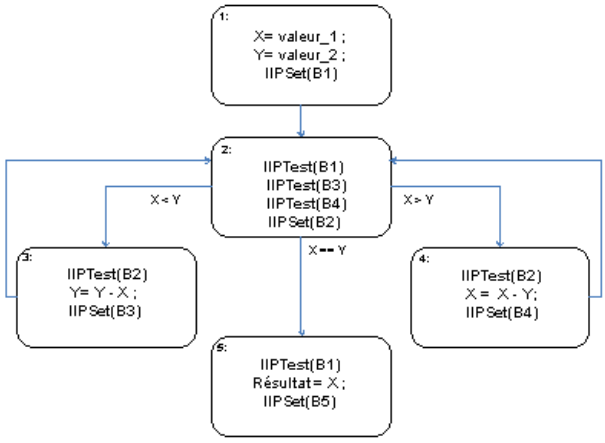


Figure 1-12 : Principe de la méthode de [Bern. 05]

La méthode proposée dans **[Bern. 05]** n'est pas capable de détecter les branchements illégaux à partir de la fin d'un bloc vers le milieu du bloc suivant. Et comme pour la méthode CFCSS, il existe quelques cas de branchements illégaux à partir du milieu d'un bloc Vi vers le milieu du

bloc Vj suivant qui ne sont pas détectés, dans le cas où le successeur de Vj est également un successeur de Vi.
Cette technique a été améliorée dans **[Bern. 06]**. En effet cette approche a été combinée avec une solution de vérification des données pour une meilleure couverture des erreurs transitoires.

I.4.2.1.3. Surveillance continue des signatures

Les techniques que nous avons vues jusqu'ici se basent sur des signatures dites "verticales". Les méthodes les plus classiques (et les plus anciennes) de vérification de flot de contrôle consistent en effet à décomposer le code en blocs et à insérer une signature de référence, le plus souvent polynomiale, au début ou à la fin de chaque bloc. Dans certains cas, des bits indicateurs peuvent être ajoutés pour permettre à un watchdog de différencier les instructions du programme des signatures. Ou encore, comme dans la méthode PSAb (Path Signature Analysis de base) **[Namj. 82]**, une mémoire étiquette externe de deux bits est utilisée pour localiser les extrémités des blocs. Les signatures de référence se situent en tout début de bloc et sont donc localisées par l'identificateur « 01 ». Lorsque le processeur arrive à la fin du bloc (repérée par l'identificateur « 11 »), la signature est comparée avec la référence lue en début de bloc, puis elle est réinitialisée.

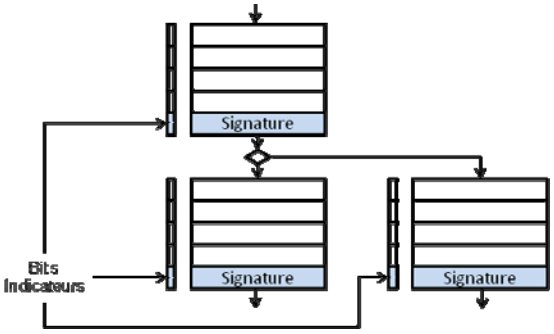


Figure 1-13 : Approche classique de vérification de flot de contrôle à base de signatures verticales

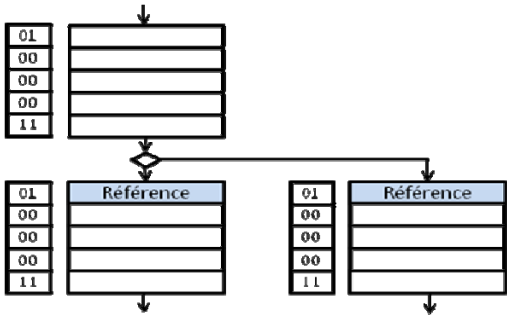


Figure 1-14 : Vérification de flot de contrôle avec PSA

L'utilisation de signatures verticales, bien qu'elle soit efficace, a le plus souvent un temps de latence très important, surtout si la vérification n'est pas effectuée à chaque fin de bloc ; sinon, c'est le coût en performances qui augmente. C'est la raison pour laquelle cette solution n'est pas suffisante quand le temps de réaction représente une contrainte majeure.

Une deuxième classe de méthodes de vérification de flot de contrôle se base sur les signatures dites "horizontales". La figure 1-15 illustre un principe de génération d'une telle signature, qui permet d'associer à chaque nouvelle instruction une nouvelle signature à vérifier. La surveillance continue de signatures ou CSM (Continuous Signature Monitoring) est une technique qui utilise à la fois les signatures verticales et horizontales afin de diminuer le temps de latence sans diminuer le taux de couvertures des erreurs. Ces signatures sont calculées et stockées pendant la phase compilation. La technique CSM est utilisée dans **[Wilk. 88] [Wilk. 90]** et **[Chen 05].**

Ces méthodes sont efficaces et garantissent un temps de latence très inférieur à celui des signatures verticales mais présentent l'inconvénient d'avoir un surcoût en mémoire très important. En effet, pour un même surcoût en mémoire, ces méthodes sont moins efficaces que les méthodes à base de signatures verticales. De plus leur efficacité est variable selon la position de l'erreur : plus le degré d'avancement dans le bloc est important, moins les erreurs sont détectables.

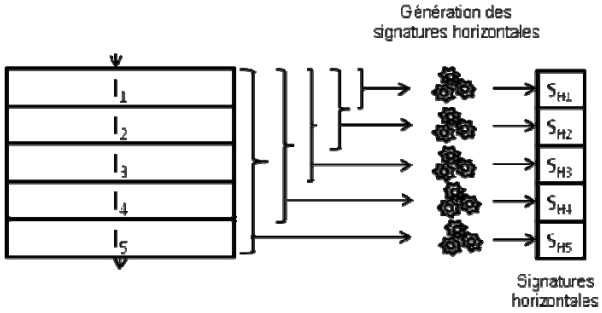


Figure 1-15 : Vérification de flot de contrôle à base de signatures horizontales

I.4.2.1.4. Conclusion sur les méthodes avec signatures enfouies

Les méthodes de vérification de flot de contrôle à base de signatures enfouies consistent à modifier le programme à vérifier en lui ajoutant des signatures de référence et parfois des instructions de calcul et de comparaison de ces signatures en ligne. Ces ajouts sont généralement effectués pendant la compilation grâce à la modification du compilateur. Néanmoins, ces modifications ne peuvent toucher que le programme à vérifier et négligent donc les appels aux fonctions externes (fonctions de bibliothèques) ainsi que les routines d'initialisation et de terminaison de l'application.

L'avantage des méthodes ESM vient du coût mémoire relativement faible pour le stockage des informations de surveillance et de la faible complexité du dispositif d'analyse de signature s'il est ajouté sous forme matérielle.

Pour toutes les méthodes ESM, il faut s'assurer que le cas où le processeur ne rencontre jamais d'instructions de test (sortie du code par exemple) est détecté. En effet, exceptées les solutions CSM **[Wilk. 88]** et **[Chen 05]**, les solutions ESM étudiées sont incapables de détecter ce type d'erreur. Le faible coût matériel du moniteur des méthodes ESM ne doit donc pas occulter le fait qu'il est nécessaire dans de nombreux cas, de compléter l'analyse de signature par un dispositif supplémentaire qui permettra éventuellement de détecter les ruptures de séquence inopinées et les sauts vers des zones mémoires non allouées au code.

I.4.2.2. Méthodes de vérification de flot de contrôle avec signatures disjointes

I.4.2.2.1. Méthode OSLC [Made. 91]

Contrairement aux approches classiques qui nécessitent une phase de prétraitement pour le calcul des données relatives aux signatures, la méthode OSLC **[Made. 91]** utilise des références calculées en ligne. En effet, OSLC se déroule en 2 phases :

- Phase d'apprentissage « Learning » :

La génération des références se fait en ligne pendant le test final de l'application. Mais pour que l'apprentissage génère toutes les références nécessaires, il faut que tous les arcs du GFC de l'application soient parcourus au moins une fois pendant cette phase. Toutefois l'utilisation d'OSLC reste possible avec une génération de références classique par un programme séparé.
Le programme est décomposé en sections associées chacune à un nombre donné de signatures de référence. Ces sections sont repérées par leurs adresses de début et adresses de fin à l'aide d'une étiquette de taille 1 bit. Ce repérage permet de détecter les ruptures de séquence inopinées ainsi que les branchements hors de la zone mémoire contenant le code.

- Phase de vérification « Checking » :

Au début de chaque bloc, la signature globale est initialisée à une valeur fixe calculée pendant la phase d'apprentissage et stockée en mémoire. A la fin du bloc, si cette signature correspond à une référence associée au segment auquel appartient l'instruction de fin du bloc, elle est considérée correcte, sinon une erreur sera détectée.
Grâce à cette méthode, le processeur de vérification ne dispose donc pas d'une image du graphe du programme vérifié, ce qui permet de réduire le coût mémoire des références. En revanche, la probabilité de détection d'une erreur dépend directement du nombre de références associées à chaque segment.

I.4.2.2.2. Méthode WDP [Mich. 93]

L'approche proposée dans **[Mich. 91]** consiste à surveiller directement les adresses du processeur principal grâce à un watchdog qui effectue deux tâches principales :

1. La détection des ruptures de séquences d'adresses ainsi que des nœuds atteints par le processeur. Ces nœuds sont classifiés en 7 catégories :
   - I_0 : Nœud d'initialisation
   - I_1 : Nœud de destination
   - I_2 : Nœud de séquencement, branchement inconditionnel
   - I_3 : Nœud de séquencement, branchement conditionnel
   - I_4 : Nœud de séquencement, branchement inconditionnel vers un sous-programme
   - I_5 : Nœud de séquencement, branchement conditionnel vers un sous-programme
   - I_6 : Nœud de séquencement, retour d'un sous-programme
2. Le calcul de la signature de la séquence d'instructions exécutée.

Le programme du watchdog contient une instruction par nœud de l'application principale.

Chaque instruction est composée de 3 champs :

- Le code opératoire : il correspond au type de nœud considéré
- L'adresse de l'instruction correspondante du programme principal
- Une référence : ce champ change de sens selon le type de nœud. En effet dans le cas d'un nœud de destination ce champ prend la valeur de la signature du nœud alors que dans le cas d'un nœud de séquencement la référence correspond au résultat de hachage (⊕) de la signature du nœud avec l'adresse dans le programme du watchdog du nœud de destination.

Lorsque le processeur arrive séquentiellement sur une destination, le watchdog compare la signature courante avec le champ de référence du nœud.
Lorsque le processeur arrive sur une instruction de séquencement, le watchdog calcule l'adresse du nœud de destination en utilisant la signature calculée pendant l'exécution et le champ de référence haché du nœud traité. Puis il charge ce nœud de destination.
Quand le processeur effectue un branchement, le watchdog compare l'adresse de la destination prise par le processeur et le champ de localisation du nœud qu'il vient de charger. En cas de différence une erreur est signalée. Puis la signature courante est réinitialisée avec le champ de référence du nœud de destination.

I.4.2.2.3. Conclusion sur les méthodes avec signatures disjointes

Comme nous l'avons vu dans le paragraphe I.4.2, les méthodes DSM ne nécessitent pas de modification du code applicatif et présentent un coût en performance théoriquement nul. L'inconvénient majeur de ces méthodes vient de leur coût mémoire. En effet, en plus des informations relatives aux signatures, le programme du watchdog doit généralement contenir des informations sur la structure du programme vérifié.

Le coût matériel est aussi en général élevé car le watchdog doit non seulement accéder à une mémoire d'une manière autonome pour ne pas dégrader les performances du système, mais aussi maintenir une synchronisation étroite avec le processeur en analysant son fonctionnement.

Pour les méthodes DSM étudiées, le cas d'un départ en exception ne présente pas un réel problème puisqu'il est identifié comme étant une erreur de séquencement. Dans ces cas là, il faut simplement que le watchdog soit capable de différencier un départ en exception d'une erreur de séquencement. Ceci peut être réalisé soit par un marquage spécial de la première instruction des routines d'exception, soit par l'ajout au début de ces routines d'une instruction spéciale qui permet d'envoyer un acquittement spécifique au watchdog, mais cette solution augmente le temps de latence et réduit légèrement les performances du processeur.

I.4.3. Synthèse sur les méthodes de vérification de flot de contrôle

En guise de conclusion à ce paragraphe, les tableaux I-I et I-II comparent les différentes méthodes CFC étudiées.

Tableau I-I: Caractéristiques générales des méthodes étudiées

	DSM/ ESM	Vérifications imposées	Taux de couverture	Temps de latence	Coût		
					Perf.	Mém.	Surf.
[Made. 91]	DSM	Blocs linéaires	91,3-99,6%	19,8-511 (µsec)	NC	NC	NC
[Mich. 91]	DSM	Blocs linéaires	99,97%	NC	0	24-61%	NC
[Oh 02]	ESM	Blocs linéaires	95,4-97,2%	NC	16-69%	25-43%	0
[Alkh. 99]	ESM	Blocs linéaires	99,8%	NC	32-138%	5%	0
[Vemu 06]	ESM	Blocs linéaires	99,8-98,9%	NC	3.15-57.8%	25-100%	0
[Vemu 08]	ESM	Super nœuds fréquents	98-99.2%	NC	28-42%	20-30%	0
[Bern. 05]	ESM	Blocs linéaires	95-100%	NC	16-69%	25-34%	3,2%
[Wilk. 88]	ESM	Permanente	99,99%	1,2-1,6 (cycles)	0,6-1,5%	3-7%	NC
[Chen 05]	ESM	Permanente	99,6-100%	0,65-0,83 (cycles)	6-11,25%	3,9-13,4%	9,6%

Tableau I-II: Erreurs détectées par les méthodes étudiées

	Erreurs sur le contenu des instructions	Erreurs de séquencement													
		Erreurs sur des instructions de branchement						Erreurs sur des instructions de non branchement					Traitement des exceptions		Traitement des sous-programmes
		A	B	C	D	E	F	A′	B′	C′	D′	E′	Départs en except.	Retour des except.	
[Made. 91]	-	-	+	+	+	+	+	+	-	-	+	+	+	+	+
[Mich. 91]	+	-	+	+	+	+	+	+	+	+	+	+	+	+	+
[Oh 02]	-	-	+	+/-	+/-	+/-	-	+	+	-	-	-	-	+	+
[Alkh. 99]	-	-	+	+	+	+	-	+	-	-	+	-	-	+	+
[Vemu 06]	-	-	+	+	+	+	-	+	+	-	+	-	-	+	+
[Vemu 08]	-	-	+/-	+/-	+/-	+/-	-	+/-	+/-	-	+/-	-	-	+	+
[Bern. 05]	-	-	+	+	+	-	-	-	-	-	+	-	-	+	+
[Bern. 06]	+	-	+	+	+	-	-	-	-	-	+	-	-	+	+
[Wilk. 88]	+	-	+	+	+	+	+	+	+	+	+	+	+	+	+
[Chen 05]	+	-	+	+	+	+	+	+	+	+	+	+	+	+	+

Pour être conforme à des exigences classiques, comme celles de la norme IEC 61508, nous allons dans ce qui suit nous intéresser surtout aux méthodes DSM. Le surcoût en mémoire n'est pas le

point le plus critique, car les systèmes intégrés modernes peuvent contenir de grandes quantités de mémoire à faible coût (voire même, disposer souvent de plus de mémoire que nécessaire pour l'application, par exemple dans des FPGA récents). Une personnalisation de la méthode WDP est très envisageable vu qu'elle répond à nos objectifs de séparation entre l'application à tester et le code de test. Il est toutefois nécessaire d'analyser l'adaptation possible de cette méthode sur des processeurs récents (notamment architectures RISC) et de chercher à améliorer son efficacité en termes de couverture globale et de latence. Pour cela, les idées utilisées dans d'autres approches étudiées, comme par exemple l'utilisation de signatures horizontales, pourront être mises à profit. La protection des données est aussi un point important à prendre en considération ; ce point n'était pas abordé dans la méthode WDP.

I.5. Identification des données critiques

Les données manipulées par un programme peuvent être altérées pendant leur stockage en mémoire. Pour détecter les erreurs qui peuvent les frapper, il est possible de les dupliquer comme nous l'avons vu dans le paragraphe I.1.3. mais le surcoût mémoire est alors très important et la comparaison systématique réduit fortement les performances. Il est donc plus judicieux de sélectionner les variables les plus critiques et de ne vérifier que celles-ci.

La détermination des variables critiques passe par la définition et le calcul de critères de criticité. Ces derniers diffèrent d'une approche à une autre mais certains sont assez récurrents comme la durée de vie (« lifetime ») et le nombre de descendants (« fanout »). Cependant la manière de les calculer n'est pas toujours la même. Nous allons donc résumer ici les principales approches proposées dans la littérature et leurs limitations, que nous chercherons à réduire dans le chapitre suivant.

Une fois les variables jugées critiques identifiées, celles-ci peuvent être par exemple dupliquées dans le programme ou dans une mémoire auxiliaire.

I.5.1. Méthode RECCO [Bens. 00]

Dans l'approche RECCO, une métrique Poids-fiabilité D_v "reliability weight" est calculée pour chaque variable en fonction des deux paramètres suivants : la durée de vie de la variable et ses dépendances fonctionnelles.
Pour calculer la durée de vie Dv(v) de chaque variable v, Benso et al. comptent le nombre de lignes de code qui séparent une instruction d'écriture de la dernière instruction de lecture de la même variable ou de la fin de l'exécution du programme. Cette méthode ne prend donc en compte ni la récursivité ni les itérations. Elle néglige également le nombre de cycles nécessaires pour exécuter une instruction donnée en fonction de l'architecture cible et des aléas dus par exemple aux dépendances de données ou aux remplissages de caches.

Le second critère utilisé dans cette méthode est les dépendances fonctionnelles D_f. La méthode utilisée pour calculer ce critère consiste à construire un graphe de dépendance (Variable Dependency Graph - VDG), dont chaque nœud représente une variable et chaque arc

représente la dépendance entre les deux variables. Un arc vi → vj existe si et seulement si vj est une descendante de vi. Ce graphe est utilisé pour calculer un facteur D_f (v).
Il suffit ensuite d'appliquer la formule suivante pour déterminer le paramètre Poids-fiabilité(v) :

$$\text{Poids - Fiabilité}(v) = Kl * Dv(v) + Kw * \Sigma \text{Poids - Fiabilité}(\text{descendants}(v))$$

Où « Kl » et « Kw » sont des coefficients de normalisation.

Prenons l'exemple d'une permutation entre deux variables (Figure 1-16): Les 3 variables sont inter-dépendantes et il est impossible de conclure si "a" dépend de "b" ou bien "b" dépend de "a". De ce fait, la formule (1) ne peut être utilisée dans ce genre de situation. Pour résoudre ce problème nous pouvons toujours réduire le graphe pour éliminer les cycles mais cette solution est irréversible et engendre une perte d'informations pour les calculs à venir.

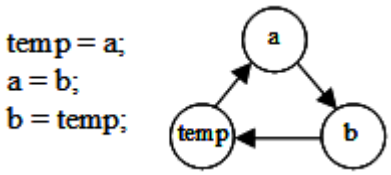


Figure 1-16 : Exemple de VDG

I.5.2. Métriques de criticité pour le placement stratégique de détecteurs [Patt. 09]

Pour représenter le plus fidèlement possible les dépendances entre les variables, la méthode de **[Patt. 09]** se base sur la construction d'un graphe de dépendance (Direct Dependency Graph - DDG) à partir du code assembleur et des scenarii d'exécution.

Le DDG est un graphe direct et acyclique (contrairement au VDG présenté au paragraphe 1.5.1.1). Il capture les dépendances dynamiques entre les différentes valeurs produites pendant l'exécution d'un programme.

Dans les DDG, chaque affectation dynamique d'une variable est traitée comme étant une nouvelle valeur. Une valeur est donc toute variable ou toute adresse mémoire utilisée dans le programme pendant son exécution. Si une variable est réécrite, elle est considérée comme étant une nouvelle valeur. Cette description résout le problème d'interdépendance relevé dans les VDG mais implique un grand surcoût mémoire.

En se basant sur ce DDG, Pattabiraman et al. peuvent calculer plusieurs critères de criticité. En effet, ils ont utilisé deux critères classiques, la durée de vie ainsi que le fanout, mais ils ont de plus défini de nouveaux critères à savoir l'exécution, la propagation et la couverture :

- La durée de vie est la distance maximale entre un nœud et son successeur immédiat en termes d'instructions dynamiques,
- Le fanout d'un nœud est l'ensemble de tous les successeurs immédiats de ce nœud dans le DDG,

- L'exécution est la fréquence d'exécution des instructions statiques associées à des nœuds,
- Le taux de propagation d'une erreur pour un nœud donné est le nombre de nœuds qui peuvent être affectés par cette erreur avant de causer un crash,
- La couverture est le nombre de nœuds à partir desquels une erreur peut se propager vers un nœud donné avant de causer un crash.

Les principales limitations de cette méthode résident dans la nécessité d'une exécution de l'application pour générer le DDG, et la dépendance potentielle de celui-ci vis-à-vis des valeurs d'entrée utilisées lors de cette exécution. D'autres limitations peuvent être citées au sujet de l'article : le fait que pour déterminer les variables critiques, une seule métrique est utilisée à la fois (absence de formule qui regroupe plusieurs critères de criticité), en plus de l'absence de méthodes automatisées pour calculer ces différentes métriques.

I.5.3. Analyse statique pour l'atténuation des erreurs logicielles dans le banc de registres [Lee 09][Lee 11]

[Lee 09] propose une analyse statique pour estimer la vulnérabilité du banc de registres afin d'atténuer les erreurs logicielles. La vulnérabilité (Register File Vulnerability) est une sorte de calcul de durée de vie, difficile à réaliser, car il dépend des chemins suivis pendant l'exécution. Cette dépendance est analysée au niveau des blocs, ce qui permet de décomposer la vulnérabilité d'un bloc entre vulnérabilité intrinsèque v_i et vulnérabilité conditionnelle v_c, v_i étant la moyenne des longueurs des intervalles fermés dans le bloc [écriture-lecture d'une variable], et v_c la longueur moyenne des derniers intervalles non fermés. Combinant ces deux paramètres avec les probabilités post-condition S, le taux du RFV est obtenu par la formule suivante :

$$\sum_j f_j V_j = \sum_j f_j (v_{ij} + v_{cj}\ S_j)$$

où f_j et V_j sont respectivement la fréquence d'exécution et la vulnérabilité du $j^{ième}$ bloc.

Une fois les registres critiques (ou vulnérables) identifiés, ils doivent être sécurisés. Les approches classiques de sécurisation des registres, sélectionnent les registres les plus critiques et ensuite les sécurisent avec des solutions logicielles et/ou matérielles comme la redondance, la parité… L'approche proposée dans **[Lee 10]** procède différemment : elle sécurise de manière matérielle les K registres du banc de registres ayant les numéros les plus grands. Ensuite elle déplace (swapping) les registres les plus critiques de l'application dans les K registres sécurisés par une phase de réallocation des registres.

Le choix d'une solution hybride permet de minimiser les coûts en énergie consommée. Néanmoins, la solution est très liée au jeu d'instructions de l'architecture cible puisque la phase de réallocation se fait sur un exécutable.

I.5.4. Calcul de l' « Importance » des variables [Leek. 10]

Trois nouvelles métriques de calcul de criticité des variables ont été présentées dans **[Leek. 10]** : l'impact spatial, l'impact temporel et l'importance, la dernière étant obtenue par la fusion des deux premières :

- L'impact spatial d'une variable « v » représente la propagation de l'erreur dans les fonctions adjacentes si « v » est corrompue,
- L'impact temporel de « v » indique la durée maximale de l'exécution pendant laquelle le programme est affecté si « v » est corrompue.

Une fois les calculs faits, deux seuils d'importance sont définis en fonction des résultats trouvés : λ_d seuil d'importance des variables à dupliquer et λ_t seuil d'importance des variables à tripliquer **[Leek. 11]**.

I.6. Approches de vérification combinée - Données et flot de contrôle

I.6.1. Exemples de méthodes de vérification combinée

I.6.1.1. Détection des erreurs de données et de flot de contrôle à l'aide d'un watchdog [Bens. 03]

L'approche proposée dans **[Bens. 03]** permet de vérifier à la fois le flot de contrôle et les données. Elle utilise la méthode RECCO décrite dans le paragraphe 1.5.1 et définie dans **[Bens. 00]**, pour déterminer et vérifier les variables critiques.
Les erreurs de flot de contrôle ont été classées en deux catégories :

- Les instructions dans un même bloc ne sont pas exécutées correctement,
- Les branchements ne sont pas exécutés dans le bon ordre.

Le premier problème peut être résolu en calculant une signature pour chaque bloc à partir des codes des opérations des instructions de ce bloc. Une signature globale est calculée hors ligne et stockée dans la mémoire interne du watchdog. Pendant l'exécution le watchdog recalcule la signature de chaque bloc et la compare avec la signature globale.

Le deuxième problème a déjà été résolu dans **[Bens. 01]** avec l'adoption d'une méthode ESM à base d'expressions régulières : à chaque bloc du programme est associé un label unique. En utilisant cette notation, l'exécution correcte du programme génère un langage composé de toutes les chaines obtenues à partir de la concaténation des symboles correspondant aux blocs parcourus. Une exécution est considérée comme correcte si le chemin suivi est correct par rapport au langage.
Pour **[Bens. 01]** la majorité des erreurs de type branchement inter-blocs sont détectées sauf les cas suivants :

- Tous les cas de branchements à partir de la fin d'un bloc vers le milieu du bloc suivant ne sont pas détectés (1)

• Quelques cas de branchement illégaux à partir du milieu d'un bloc vers le milieu ou la fin d'un autre bloc ne sont pas détectés : dans le cas d'un branchement conditionnel, tant que l'instruction « write » du bloc correspondant au prédicat P n'a pas encore été effectuée, il peut y avoir un branchement illégal vers le début (2a) ou le milieu (2b) du bloc correspondant au prédicat P.

Tableau I-III: Erreurs détectées par les méthodes [Bens. 01] et [Bens. 03]

	Erreurs sur le contenu des instructions	Erreurs de séquencement													
		Erreurs sur des instructions de branchement						Erreurs sur des instructions de non branchement					Traitement des exceptions		Traitement des sous-programmes
		A	B	C	D	E	F	A'	B'	C'	D'	E'	Départs en except.	Retour des except.	
[Bens. 03]	+	-	+	+	+	NC	-	NC	-	-	NC	-	-	+	+
[Bens. 01]	-	-	+	+	+	-	-	+/-	-	-	+/-	-	-	+	+

Tableau I-IV: Caractéristiques générales des méthodes [Bens. 01] et [Bens. 03]

	DSM/ ESM	Vérifications imposées	Taux de couverture	Temps de latence	Coût		
					Perf.	Mém.	Surf.
[Bens. 03]	DSM	Blocs linéaires – E/S mémoire	30-70%	NC	60%	NC	NC
[Bens. 01]	ESM	Blocs linéaires	24-81%	NC	116,66%-300%	8,33%-21,5%	0

I.6.1.2. Détection des erreurs d'accès aux données [Wilk. 97]

[Wilk. 93] propose une approche de vérification des accès mémoire permettant de détecter les erreurs logicielles et certaines erreurs matérielles. La technique proposée crée une redondance dans les structures de données en ajoutant une signature dans chaque instance des structures (Data Structure Signature - DSS). Le DSS est une constante définie à la création de la structure et placée dans le mot qui précède le premier élément de la structure. La figure 1-17 illustre un exemple de structure de données redondante.

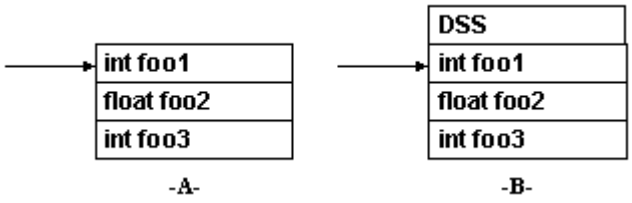


Figure 1-17 : Transformation d'une structure de données (4-A : Structure classique, 4-B : Structure redondante)

Dans une structure redondante, le pointeur sur la structure continue de pointer sur le premier élément, ceci permet au programme de continuer d'accéder aux éléments de la structure de la manière habituelle.

Le DSS exploite la redondance spatiale pour la détection des accès mémoire erronés de la manière suivante : pour un pointeur p qui pointe sur une structure de données, le compilateur connait le décalage (égal à -1) entre le pointeur et DSS. Ainsi, le compilateur peut insérer des instructions qui vérifient le pointeur p par chargement de la valeur à l'emplacement p-1, et en comparant cette valeur avec la signature de référence pour cette structure de données. Si une erreur s'est produite, la vérification des DSS d'exécution permet de la détecter.

[Wilk. 97] mixe la redondance des structures de données, basée sur les DSS, et la vérification de flot de contrôle avec signatures de justification **[Namj. 82]**. En effet, cette vérification permet de surveiller les DSS en même temps que les signatures de flot de contrôle.
La méthode utilise les variations dans les signatures pour détecter les erreurs. La figure 1-18-A montre la technique basique de CFC. Afin de réduire le surcoût en mémoire les signatures d'ajustement sont utilisées comme illustré dans la figure 1-18-B. Celles-ci permettent de garder la cohérence dans le calcul de signature quand deux chemins différents peuvent être pris.

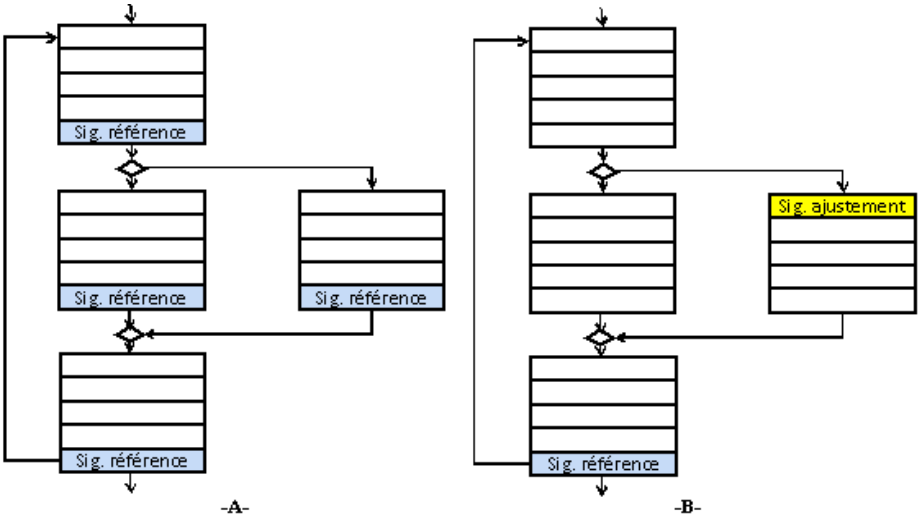


Figure 1-18 : Placement des signatures dans un programme (18-A Technique CFC classique, 18-B Technique CFC avec des signatures d'ajustement)

La mise à jour de la signature se fait avec la formule suivante : $S_{i+1} = S_i\alpha \oplus I_i \oplus DSS_i$, où $S\alpha$ est la signature courante et I est le code machine de l'instruction de chargement du DSS. Pour les autres instructions, la signature est mise à jour avec cette formule : $S_{i+1} = S_i\alpha \oplus I_i$. .

Tableau I-V: Erreurs détectées par la méthode [Wilk. 97]

	Erreurs sur le contenu des instructions	Erreurs de séquencement													
		Erreurs sur des instructions de branchement						Erreurs sur des instructions de non branchement					Traitement des exceptions		Traitement des sous programme
		A	B	C	D	E	F	A′	B′	C′	D′	E′	Départs en except.	Retour des except.	
[Wilk. 97]	+	-	+	+	+	+	-	+	+	+	+	-	NC	NC	NC

Tableau I-VI: Caractéristiques générales de la méthode [Wilk. 97]

	DSM/ ESM	Vérifications imposées	Taux de couverture	Temps de latence	Coût		
					Perf.	Mém.	Surf.
[Wilk. 97]	ESM	Blocs linéaires	65% - 100%	NC	0% -13%	0%-120%	NC

I.6.2. Synthèse sur les méthodes de vérification combinée

Plusieurs méthodes de vérification combinée sont présentes dans la littérature, mais aucune d'elles ne présente un taux de couverture maximal à cause des lacunes dans leurs approches de vérification de flot de contrôle.

I.7. Conclusion

Nous avons vu dans ce chapitre que les problèmes liés à la sûreté de fonctionnement sont multiples et croissants et que les sources d'erreurs augmentent avec la réduction des dimensions des procédés microélectroniques. Les fautes transitoires doivent désormais être prises en compte au niveau du sol et plus seulement pour les applications spatiales. Le caractère multiple des fautes ne peut plus être ignoré comme par le passé.

Concernant les techniques de protection, nous avons vu qu'elles peuvent être basées sur du matériel, sur du logiciel ou sur une collaboration entre matériel et logiciel. Les erreurs observées étant de plus en plus souvent des erreurs multiples, les protections classiques à base de codes détecteurs deviennent soit inefficaces, soit trop coûteuses, d'où notre intérêt pour les méthodes de vérification de flot de contrôle.

Par la suite, nous avons survolé les méthodes de vérification de flot de contrôle et de données afin de les comparer selon plusieurs critères tels que le taux de couverture des erreurs, le temps de latence ou encore les surcoûts en terme de mémoire, performance et surface. Les différentes méthodes publiées négligent les appels aux fonctions externes ainsi que les routines d'initialisation et de terminaison de l'application. Par ailleurs, aucune ne permet d'atteindre un taux de couverture satisfaisant à la fois vis-à-vis des erreurs de flot de contrôle et vis-à-vis des erreurs de données.

Dans le chapitre II, nous allons présenter une nouvelle méthode de vérification de flot de contrôle palliant la plupart des limitations identifiées dans les méthodes existantes, et étendue pour couvrir aussi les erreurs dans les données critiques. Ceci nécessitera par ailleurs de

revisiter les méthodes d'évaluation de criticité des variables, en tenant compte des limitations identifiées dans les méthodes existantes. Essentiellement pour des soucis de conformité aux normes, telles que la norme IEC61508 qui porte sur les systèmes électroniques et électriques de sécurité **[Mari. 06][Mari. 07]**, la méthode proposée sera une méthode DSM.

Chapitre II :

Présentation de la méthode IDSM

Dans ce chapitre nous allons présenter une nouvelle méthode de vérification de flot de contrôle, nommée IDSM « Improved Disjoint Signature Monitoring ». Cette méthode doit répondre à cinq objectifs principaux :

1. Taux de couverture et latence : elle doit couvrir le maximum de types d'erreurs de flot de contrôle et de données, et la latence de détection doit être faible,
2. Séparation des éléments initiaux et des éléments de surveillance ajoutés au système vérifié, pour assurer la compatibilité de l'approche avec certaines normes ; elle doit donc appartenir à la famille des méthodes DSM,
3. Généralité : la méthode proposée doit être assez générique pour qu'elle soit applicable avec la plupart des processeurs,
4. Performances : la surveillance continue du comportement du système ne doit pas induire de ralentissement de l'application par rapport au système sans surveillance,
5. Limitation des coûts : le coût de mise en œuvre, le matériel ajouté et le coût mémoire doivent être réduits autant que possible, afin de garder un avantage net par rapport à une simple duplication.

Dans le paragraphe II.1 nous présenterons d'une manière générale la méthode proposée. Dans les paragraphes II.2 et II.3 nous aborderons les problèmes techniques liés aux caractéristiques du processeur et du programme à vérifier (les routines d'initialisation et de terminaison, ainsi que les fonctions de bibliothèques). Les paragraphes II.4, II.5 et II.6 feront l'objet de la description succincte du jeu d'instructions du watchdog, son architecture ainsi que son comportement général. Et enfin, le paragraphe II.7 est consacré à une discussion sur la couverture effective des erreurs de la méthode IDSM. Dans ce chapitre, la description de l'approche reste à un niveau générique. Le chapitre III donnera davantage de détails pour une étude de cas basée sur un processeur Sparc v8.

II.1. L'approche IDSM

II.1.1. Principes généraux de la méthode proposée

Pour sécuriser un système avec IDSM, nous devons y inclure notre co-processeur de surveillance (ou « watchdog ») afin qu'il puisse surveiller le flux circulant entre le processeur et la mémoire et plus précisément entre l'unité d'exécution et la mémoire cache (pour des raisons que nous expliquerons dans le paragraphe II.2.1). Le watchdog va surveiller ce qui passe sur les bus à chaque cycle, la vérification du bon comportement du système étant assurée par des informations sur le GFC. En effet, comme pour la plupart des méthodes DSM, le watchdog doit disposer d'informations sur le squelette du programme à vérifier. Ces informations forment un programme de vérification, accédé par un pointeur et exécuté par le watchdog. Le but d'IDSM

étant de détecter et signaler les erreurs, et non pas de les corriger, le watchdog doit envoyer un signal d'erreur vers le processeur lui-même ou vers un élément superviseur implanté dans le système en cas de saut illégal ou de corruption de données. Un schéma possible pour les interconnexions des éléments principaux est illustré dans la figure 2-1.

Le repérage des instructions à l'origine ou à la destination d'une rupture de séquence du programme vérifié est fait par leur adresse, comme pour la méthode WDP. L'avantage de ce type de repérage est de ne pas nécessiter le décodage des instructions du processeur vérifié. Il peut donc s'appliquer d'une manière générale, quel que soit le processeur.

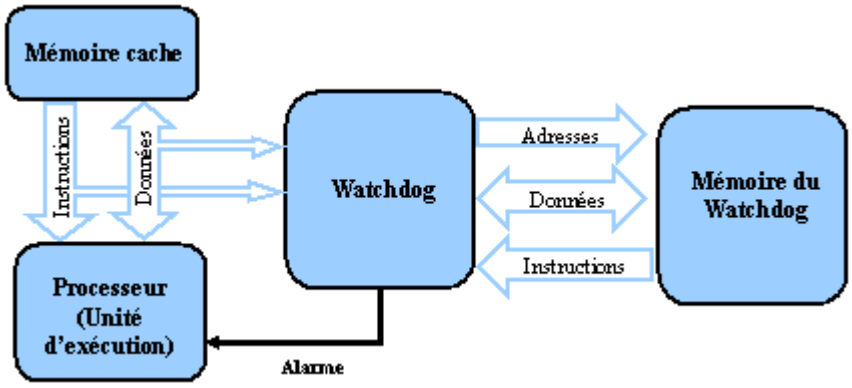


Figure 2-1 : Architecture d'un système sécurisé avec IDSM

Le watchdog doit également gérer un compteur de programme pour pouvoir imiter le processeur à vérifier (sans pour autant bien sûr réaliser toutes les opérations exécutées par le processeur surveillé). L'idée est de placer une instruction watchdog en correspondance avec chaque instruction à l'origine ou à la destination d'une rupture de séquence du programme vérifié, c'est-à-dire à chaque extrémité des blocs linéaires. Dans ce qui suit nous appellerons singularité toute instruction extrémité de bloc. Ainsi le watchdog connaît par moyen externe, l'emplacement de toutes les instructions de séquencement dans le programme vérifié et il peut donc détecter les ruptures de séquence inopinées.
Les informations associées à ces instructions sont, outre leur adresse dans le programme vérifié, le type de singularité et une valeur de référence permettant d'effectuer l'analyse de la signature. Le type de singularité correspond au code opératoire d'une instruction watchdog. Toutes ces informations sont stockées dans une mémoire séparée du programme et dédiée au watchdog. Le fait d'avoir une mémoire dédiée permet l'accès aux informations pendant la vérification du flot de contrôle, parallèlement aux accès effectués par le processeur dans ses mémoires.

Fonctionnement sur les instructions intra-bloc :
Pour chaque instruction intra-bloc exécutée, le watchdog compacte cette instruction et vérifie s'il n'y a pas eu de saut illégal. Pour cela, et pour toutes les instructions (sauf celle de destination), il compare l'adresse relative de l'instruction avec celle de l'instruction précédente. Dans le cas où la différence entre ces deux adresses est supérieure à la taille d'une instruction, le watchdog reconnaît une rupture de séquence inopinée.

Fonctionnement sur les instructions de saut inconditionnel :
Quand le processeur arrive sur une instruction de saut inconditionnel, le watchdog doit le suivre et charger l'instruction de destination dans son programme de vérification. Une fois ce nœud chargé, il vérifie que le processeur ait pris la bonne instruction de destination.

Fonctionnement sur les instructions de saut conditionnel :
Le fonctionnement pour des instructions de saut conditionnel est différent car le branchement peut être pris ou pas en fonction des drapeaux (ou bits d'état) du processeur.
Idéalement, le watchdog permettra de vérifier non seulement la légalité du chemin suivi, mais aussi sa correction. L'idée serait d'évaluer directement les conditions de branchement, et pour ce faire, deux solutions sont possibles :

1. Le watchdog accède au registre d'état du processeur et détermine en fonction de son contenu le chemin qui doit être suivi. Dans le cas d'une méthode DSM, il n'est pas possible d'accéder directement au registre d'état du processeur. Il faudrait donc non seulement décoder les instructions de branchement pour savoir quel drapeau doit être testé mais il faudrait aussi avoir une information locale sur sa valeur.
   - ⇨ Cette méthode, certes potentiellement efficace, oblige le watchdog à décoder les instructions de branchement et à avoir un comportement spécifique pour chaque type de saut. Il faudrait en plus pouvoir effectuer presque les mêmes calculs que le processeur pour garder une trace de la valeur du registre d'état. Elle est donc très coûteuse en performance, en matériel et en mémoire. C'est pour ces raisons là qu'elle ne peut pas être retenue.
2. Chaque branchement conditionnel est généralement précédé par une instruction I_{cond} qui va modifier les drapeaux et en fonction de laquelle le branchement sera effectué ou pas. L'idée est de repérer ces instructions et de les copier dans la mémoire du watchdog pendant la phase de génération du programme de vérification.
   Pendant le test en ligne, quand le watchdog arrive sur une instruction de saut conditionnel, il pointe sur l'instruction I_{cond} correspondante dans sa mémoire interne et évalue la condition du branchement. En fonction du résultat obtenu, le watchdog charge le nœud de destination correspondant dans son programme. Une fois le nœud de destination chargé, le watchdog compare l'adresse de l'instruction prise par le processeur avec celle stockée dans le nœud destination chargé.
   - ⇨ Cette solution est également à rejeter car d'une part elle suppose que le watchdog a accès aux valeurs mises à jour des variables et d'autre part elle induit une complexité conséquente du watchdog et de ce fait engendrerait des coûts importants.

Compte tenu des coûts de mise en œuvre que nécessite la vérification de la correction des chemins, ce type d'erreurs ne sera pas pris en compte dans la méthode proposée, comme dans les autres méthodes présentées précédemment.

Lorsque le processeur arrive sur une instruction de branchement conditionnel, le watchdog devra donc attendre pour voir si le processeur va effectuer le branchement ou pas :

- Si le branchement est effectué, le watchdog charge le nœud de destination et une fois ce nœud chargé, il compare son adresse avec celle du nœud pris par le processeur.
- Si le branchement n'est pas effectué, le watchdog charge le nœud situé en séquence dans son programme de vérification et attend que le processeur arrive sur une autre singularité.

Traitement des appels aux sous-programmes :
Les sous-programmes sont généralement utilisés dans le but d'être réutilisés plusieurs fois dans le même programme et cela pour économiser en nombre de lignes de code. A partir de là, il est clair qu'un sous-programme peut être appelé plus d'une fois pendant l'exécution du programme et a de ce fait des origines multiples. Les instructions de retour de sous-programme sont donc des cas particuliers d'instructions de saut puisqu'elles peuvent avoir plus de 2 destinations. C'est pour cela qu'il est nécessaire de les traiter à part :

- Fonctionnement sur les instructions d'appel au sous-programme : le watchdog se comporte presque de la même manière que dans les cas de saut inconditionnel. La seule différence est qu'il devra stocker dans une pile l'adresse de l'instruction d'appel ainsi que la signature intermédiaire.

- Fonctionnement sur les instructions de retour d'un sous-programme : une fois le sous-programme terminé, le watchdog peut enchainer sa vérification en revenant à l'adresse stockée dans la pile et en restaurant la valeur de la signature intermédiaire.

Bien évidemment, le watchdog devra également vérifier qu'il n'y a pas eu de saut illégal pendant l'exécution du sous programme. Pour cela, il devra réinitialiser la signature au moment de l'appel au sous-programme et procéder exactement comme pour les autres blocs (compaction des instructions et test de la signature pour chaque singularité trouvée).

Traitement des exceptions :
Comme nous l'avons vu dans le paragraphe I.4.1.3, le traitement des exceptions passe par trois étapes : la reconnaissance du départ d'exception, le traitement associé au départ en exception et le traitement associé au retour d'exception.

1. Reconnaissance du départ en exception :
   La méthode proposée, tout comme la méthode WDP, possède un mécanisme de détection des ruptures de séquence inopinée et peut de ce fait reconnaitre instantanément le départ en exception. Tout ce qui reste à faire est de pouvoir différencier un départ en exception d'une vraie erreur de flot de contrôle.
   La technique utilisée dans WDP, pour ne pas confondre une routine d'exception avec un saut illégal, consiste à marquer les débuts des routines d'exception avec une instruction

spéciale. Le rôle de cette instruction est également de fournir au watchdog un pointeur sur le début du programme de vérification associé à la routine d'exception.

➔ Cette méthode nous obligerait donc à modifier le code de l'application principale, ce que nous cherchons à éviter.

Pour reconnaître les départs en exception sans toucher à l'application, notre watchdog devra détecter les accès du processeur au vecteur d'exceptions. En effet, quand une exception est déclenchée, la grande majorité des microprocesseurs accèdent à un vecteur d'exceptions pour trouver l'adresse de la routine adéquate à exécuter.

2. Traitement associé au départ en exception :
   Une fois l'exception reconnue, le watchdog empile la signature courante pour être en mesure de reprendre le calcul de la signature lors du retour d'exception. Ensuite, il initialise le dispositif de compaction afin de pouvoir vérifier le flot de contrôle de la routine d'exception.
   Il faut également assurer la cohérence du pointeur de programme. Ceci est réalisé en plaçant le pointeur dans une pile au moment du départ.

3. Traitement associé au retour d'exception :
   La reconnaissance du retour de la routine d'exception ne pose aucun problème. Le watchdog devra d'abord tester la signature de la routine d'exception ensuite recharger le dispositif de compaction avec la signature empilée au moment du départ.
   Le watchdog devra également restaurer son pointeur de programme et charger le nœud qui lui correspond dans son application de vérification.

II.1.2. Identification et vérification des variables critiques

II.1.2.1. Identification des variables critiques - Critères de criticité

Pour identifier les variables à vérifier, nous calculons pour chaque variable son taux de criticité en fonction des trois critères : sa durée de vie, ses dépendances fonctionnelles et son taux de participation dans les conditions de branchement. Le critère « fanout » communément utilisé dans les méthodes de l'état de l'art, ne sera pas utilisé dans notre formule de calcul de criticité étant donné que notre critère de dépendance entre les variables permet d'englober le fanout.

II.1.2.1.1. Durée de vie des variables

Le premier critère de criticité adopté est la durée de vie des variables C_l. En effet, les données les plus susceptibles de corruption sont celles stockées dans des variables qui ont une longue durée de vie vu qu'elles sont stockées en mémoire pour une période plus longue.

II.1.2.1.2. Dépendances entre les variables

Le second critère de criticité est relié aux dépendances fonctionnelles entre les variables. Ce critère a déjà été utilisé par le passé mais les techniques utilisées pour le calculer étaient généralement soit non optimisées soit très coûteuses. Nous proposons donc une nouvelle méthode de calcul pour évaluer les dépendances fonctionnelles entre les variables d'une manière précise et avec un surcoût mémoire faible. On définit DD(v) l'ensemble des descendants directs d'une variable "v", autrement dit toutes les variables dont la valeur utilise la valeur de v. Partant d'un ensemble DD(v), nous pouvons déduire l'ensemble de tous les descendants de "v" en utilisant la formule récursive suivante :

$$Descendant\ (v) = DD(v) \bigcup_{w \in DD(v)} Descendant\ (w)$$

La méthode proposée consiste à créer une matrice M dont les dimensions sont N*N (N étant le nombre de variables du programme) : une cellule M(a,b) > 0 indique que "a" descend de "b". L'algorithme utilisé pour créer la matrice est le suivant :

- 1ère étape : Initialisation

Cette étape permet de dérouler les instructions du programme une à une pour relever les descendants directs de chaque variable.

```
if (w ∈ DD(v)) then M0(w,v) ← M0(w,v) +1;
```

- 2ème étape : Calcul des dépendances

Cette étape a pour but de déterminer tous les descendants d'une variable autres que ses descendants directs. Tout d'abord la matrice obtenue pendant la première étape de l'algorithme est multipliée par un coefficient degree_coef pour donner plus de poids aux descendants directs. Ensuite on parcourt la liste des descendants directs et on prend en considération leurs descendants. Cette étape est répétée jusqu'à la convergence de la matrice M tout en multipliant à chaque itération la matrice par degree_coef pour donner un poids aux dépendances selon leurs degrés.

```
M1 ← M0;
Do {
    M2 ← degree_coef *M1;
    Foreach (M2(j,i) >0)
         Foreach (M2(k,j) >0)
              M2(k,i) ← M2(k,i) + M1(k,j);
    M1 ←  degree_coef *M1;
} while (!compare (M2, M1)
```

Revenons à l'exemple de permutation de variables défini dans le paragraphe 1.5.1.1. Les matrices obtenues sont représentées dans la figure 2-2. Les valeurs des cellules de M1 illustrent les interdépendances entre les 3 variables mais donnent plus de poids pour les descendances directes.

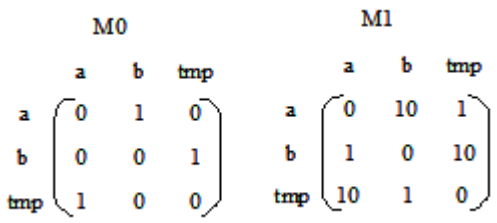


Figure 2-2 : Matrices générées pour l'exemple de permutation program - degree_coef=10

II.1.2.1.3. Participation dans les conditions de branchement

Les facteurs « durée de vie » et « dépendance fonctionnelle » ne suffisent pas pour déterminer le degré de criticité d'une variable. Prenons le cas d'un compteur « i » dans une boucle. Ce compteur peut n'avoir aucun descendant et sa durée de vie peut ne pas être longue (le temps de la boucle) mais sa valeur influe directement sur le flot de contrôle.

Intuitivement, les variables qui influent sur le flot de contrôle sont toute variable utilisée dans une instruction de branchement conditionnel. Nous pouvons donc parcourir le GFC instruction par instruction et pour chaque instruction qui a plus d'un successeur (autre que les instructions de retour de fonction ou de routine d'exception), incrémenter un facteur C_w associé aux variables qu'elle utilise.

II.1.2.1.4. Calcul du taux de criticité des variables

En utilisant la matrice de dépendance M il est possible de quantifier la criticité d'une variable « v » en utilisant la formule suivante:

$$\text{Criticité}(v) = K_l * C_l(v) + K_w * C_w(v) + K_d * \sum_{z \subset desc(v)} M(v, z) * (K_l * C_l(z) + K_w * C_w(z))$$

où C_l et C_w sont respectivement la durée de vie et le poids dans les conditions de branchement de la variable « v ». Kl, Kd et Kw sont des coefficients de régulation respectivement associés à la durée de vie, la dépendance fonctionnelle et le poids dans les conditions de branchement. Ils peuvent être utilisés pour mettre l'accent davantage sur un critère plutôt que les autres en fonction du type d'application.

II.1.2.2. Vérification des variables critiques

Pour détecter les erreurs sur les données, la solution proposée consiste à dupliquer les variables critiques du programme dans la mémoire du watchdog. Ces dernières sont caractérisées par une longue durée de vie, beaucoup de descendants et par leur participation dans des conditions de branchement. Plus généralement, une variable est considérée critique si sa criticité, calculée selon le principe présenté dans la section précédente, est supérieure à un seuil fixé pour l'application considérée.

La copie d'une variable critique est mise à jour à chaque instruction d'écriture (store) qui lui correspond. Pour chaque instruction de lecture de données de la mémoire (load), correspondant à une variable critique, le watchdog compare la valeur de la variable avec la copie stockée dans sa mémoire.

II.1.3. Vérification des instructions fréquentes

Pour diminuer le temps de latence et avoir une couverture maximale, il est intéressant de faire un suivi continu de l'exécution c'est-à-dire que pour chaque instruction exécutée le watchdog calcule une signature horizontale et la compare avec la signature stockée en interne.
Mais par soucis d'économie en mémoire et en performance cela ne sera fait que pour les instructions les plus fréquemment exécutées. Le fait d'intensifier la vérification selon le principe de localité a déjà été proposé dans [Vemu 08]. Les informations concernant la fréquence de l'exécution des instructions pourront être extraites à partir des données générées par le compilateur GCC sur les fréquences et la probabilité d'exécution des blocs du GFC de l'application. Le nouveau partitionnement du programme est illustré dans la figure 2-3.

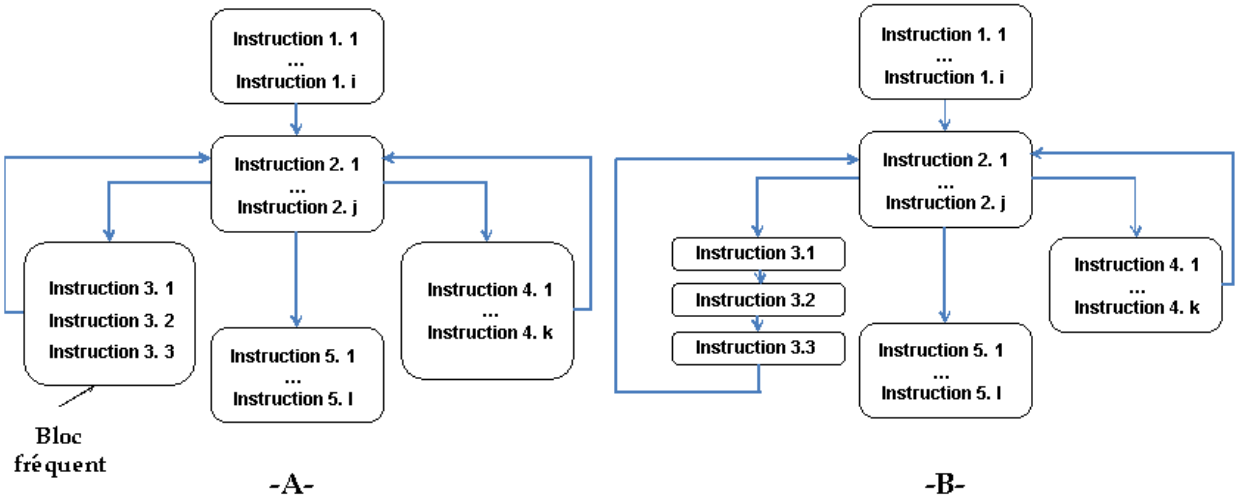


Figure 2-3 : Transformation du GFC (2-A partitionnement classique, 2-B prise en compte des blocs fréquents)

Bien évidemment, le seuil de fréquence est laissé au choix de l'utilisateur, il pourra l'augmenter et le diminuer à sa guise en fonction de ses objectifs : diminution de la latence ou du surcoût mémoire ou un compromis entre les deux.

II.1.4. Démonstration qualitative de la couverture d'erreurs

Ce paragraphe décrit le comportement du watchdog face aux types d'erreurs présentés dans le paragraphe I.4.1.4, ainsi que face aux erreurs sur les contenus des instructions.

Erreurs sur les instructions de branchement

Compte tenu des coûts de mise en œuvre que nécessite la vérification de la correction des chemins (illustré par le type A dans la figure 2-4), ce type d'erreurs n'a pas été pris en compte dans la méthode IDSM.

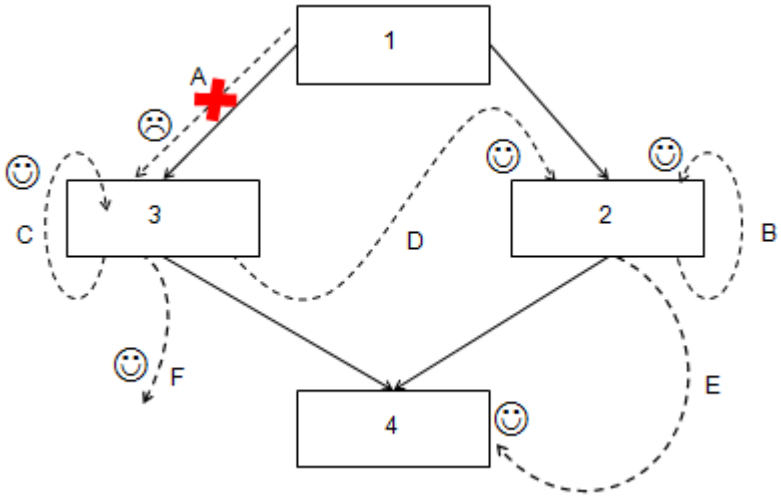


Figure 2-4 : Détection des erreurs sur les instructions de branchement

Cependant, tous les branchements illégaux dont les départs sont des instructions de branchement sont détectés.

En effet, à la fin de chaque bloc, le watchdog arrive sur une instruction de séquencement. Dans cette dernière est stockée l'adresse de destination correspondante dans le programme de vérification. Le watchdog charge alors cette instruction et compare l'adresse placée dans cette instruction avec celle prise par le processeur. Si une erreur est survenue, le résultat de comparaison permettra de la dévoiler.

Erreurs sur les instructions de non branchement

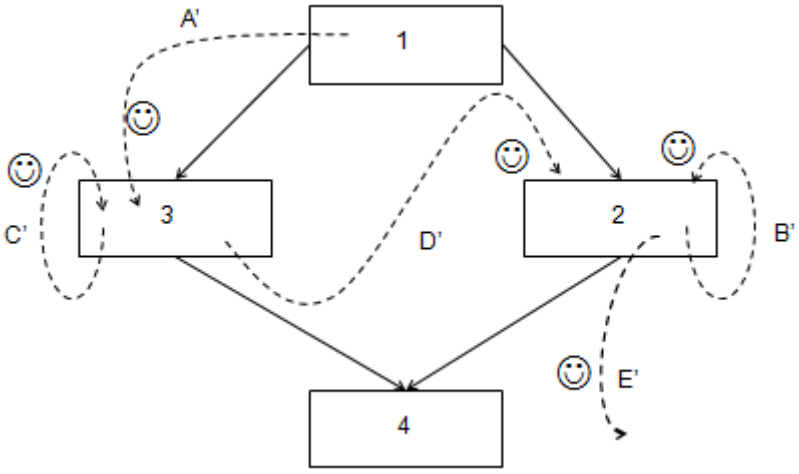


Figure 2-5 : Détection des erreurs sur les instructions de non branchement

Pour chaque instruction, le watchdog compare son adresse avec l'adresse de l'instruction précédente pour détecter les cas de rupture de séquence inopinée. Il sera donc possible de détecter toutes les erreurs sur les instructions de non branchement, comme illustré dans la figure 2-5.

Erreurs sur les contenus des instructions

Grâce à la comparaison des signatures calculées en ligne avec des signatures de références, à la fin de chaque bloc, nous pouvons nous assurer de l'intégrité des instructions exécutées. De plus, le calcul de signatures horizontales pour les instructions fréquentes réduit le temps de latence à la durée d'exécution d'une instruction et nous évite d'attendre la fin du bloc.

Tableau récapitulatif

Tableau II-I: Erreurs détectées par la méthode proposée

	Erreurs sur le contenu des instructions	Erreurs de séquencement													
		Erreurs sur des instructions de branchement						Erreurs sur des instructions de non branchement					Traitement des exceptions		Traitement des sous programme
		A	B	C	D	E	F	A′	B′	C′	D′	E′	Départs en except.	Retour des except.	
IDSM	+	-	+	+	+	+	+	+	+	+	+	+	+	+	+

II.2. Prise en compte des caractéristiques du processeur vérifié

Les microprocesseurs actuels disposent souvent de mécanismes gênants pour les méthodes de vérification de flot de contrôle. Ces mécanismes sont d'autant plus gênants dans le cas des méthodes DSM quand c'est un dispositif externe au processeur vérifié qui doit surveiller l'exécution de l'application. En effet, la méthode proposée suppose par exemple que le watchdog connaît à tout instant t quelle est l'instruction en cours d'exécution pour pouvoir calculer la signature dérivée correspondante. Or des caractéristiques comme les mémoires caches, la MMU ou encore les pipelines et la prédiction des branchements rendent l'exécution de l'application quasi-invisible par rapport aux dispositifs externes et de ce fait rendent la tâche difficile voire impossible dans certains cas.

Dans ce qui suit, nous allons préciser les contraintes d'utilisation de la méthode proposée et essayer de trouver des solutions pour contourner les effets de certains mécanismes.

II.2.1. IDSM et les mémoires cache, TCM et autres

Un des principaux compléments apportés à une architecture pipeline est l'utilisation de plusieurs niveaux de mémoire. Cela est dû au fait que les temps d'accès aux mémoires de grande taille est notablement supérieur au temps de cycle atteint par les processeurs dans les technologies modernes et donc les opérations de lecture/écriture sont devenues les vraies pierres d'achoppement des architecture modernes. Le coût des mémoires embarquées à haute vitesse a baissé, mais pas assez pour pouvoir les utiliser comme seule mémoire centrale, sans compter les contraintes technologiques et notamment la consommation. L'idée de base est donc

d'interposer entre le CPU et la Mémoire Centrale (MC), grande mais lente, une mémoire plus petite mais nettement plus rapide, appelée mémoire cache.

Quand le processeur souhaite lire une donnée à partir de la mémoire, il génère l'adresse correspondante, et émet une requête sur le bus, qui est interceptée par le cache. Si la donnée est présente dans le cache (on parle alors de « cache hit »), elle est directement envoyée au processeur. Sinon, en cas de défaut de cache (« cache miss »), la requête est transmise à la mémoire centrale. Lorsque la donnée est fournie par la mémoire, une copie est conservée (en cas d'accès futur) dans le cache, qui doit libérer la place nécessaire à son stockage. La mémoire cache améliore donc les performances du processeur mais génère néanmoins des aléas dans l'exécution à cause des cache miss.

Pour les applications critiques qui nécessitent de la performance, les TCMs (Tightly Coupled Memory) permettent de résoudre le problème des aléas spécifiques aux mémoires caches, puisque les TCMs sont remplis une seule fois au moment de l'initialisation avec les sections critiques du code, tout en offrant une latence comparable à celle des mémoires caches.

A part les mémoires caches et TCM, d'autres mémoires peuvent s'ajouter à l'architecture et le résultat sera une hiérarchie de mémoires qui deviennent généralement plus grandes et plus lentes au fur et à mesure que l'on s'éloigne du CPU.

Avec ces mécanismes, le fait de contrôler seulement les bus externes ne donne plus une idée sur le flot de contrôle. En effet, le problème ici est double :

1. Tant qu'il n' ya pas eu de cache miss les instructions présentes dans le cache peuvent ne pas s'exécuter dans l'ordre ou peuvent être altérées (cas de la corruption du PC ou de la mémoire cache) sans que cela ne soit visible par le watchdog.
2. Les instructions à exécuter peuvent provenir de plusieurs sources différentes.

Il est donc impératif pour le watchdog de voir tout ce qui transite entre chaque mémoire et le CPU, même si cela signifie qu'il faut utiliser des signaux internes du processeur, ou plus précisément avoir la capacité de se connecter sur l'ensemble des bus connectant le processeur à son premier niveau de mémoire(s). Cela ne nécessite pas à proprement parler de modifier le système initial. Toutefois, il est parfois nécessaire de rajouter des sorties à la description du processeur, pour avoir accès à certains signaux. Dans le cas d'un bloc ré-utilisable (IP), incluant le processeur et certaines mémoires, pour lequel on ne dispose que d'une description physique ("hard IP") ou d'une description protégée par chiffrement, cette contrainte peut empêcher l'application de la méthode.

II.2.2. IDSM et Unité de gestion de la mémoire

La position du code d'un programme dans le plan mémoire peut ne pas être fixe mais déterminé par un mécanisme spécial, la MMU, qui convertit les adresses logiques en adresses physiques destinées à la mémoire. Or la méthode IDSM repose sur la vérification des adresses du programme et celles-ci doivent donc être connues avant la génération du programme du

watchdog et ne doivent pas varier pendant l'exécution du programme. Pour cela, il vaut mieux que le processeur watchdog ait accès aux adresses logiques du microprocesseur et non aux adresses physiques.

Le problème de la MMU intégrée a été déjà posé dans **[Mich. 93]**. La solution proposée consiste à n'utiliser que la partie de l'adresse restant invariante lors de la transformation de l'adresse logique en adresse physique c'est-à-dire les bits de poids faible. Bien évidemment, le nombre de ces bits ne peut pas être supérieur au nombre de bits définissant la taille d'une page dans le cas d'une mémoire paginée. Dans notre méthode, nous pouvons procéder de la même manière.

II.2.3. IDSM et le pipeline

Pour un générateur de signature externe au processeur, les instructions à compacter ne peuvent être enregistrées qu'au moment de leur lecture en mémoire. Cependant, il existe des cas où une instruction est lue par le processeur sans qu'elle ne soit complètement exécutée. Ces cas sont fréquents avec une architecture pipeline, et encore plus avec une architecture utilisant une exécution spéculative lors des branchements. Il faut donc pouvoir gérer la signature en synchronisant précisément le comportement du watchdog et celui du pipeline du processeur, qui n'est pourtant pas directement visible.
Un premier problème de synchronisation se pose avec les aléas dus aux dépendances de données, qui peuvent conduire à des arrêts temporaires de la progression de certaines instructions dans les pipelines (« pipeline stall »). En l'absence d'exception, ces arrêts temporaires peuvent toutefois ne pas avoir de conséquence réelle sur la vérification de signature.
Les principaux problèmes pour la signature en présence d'un pipeline sont surtout liés à l'exécution d'un saut conditionnel ; ces complications sont appelées « aléas de contrôle ». Le drapeau qui permet au séquenceur de déterminer si un saut doit être pris ou non n'est en général généré que lorsque l'instruction précédente est dans l'étage d'exécution. L'étage de décodage de l'instruction de branchement n'est pas en mesure de calculer l'adresse de l'instruction suivante tant que la valeur du drapeau n'est pas calculée. Il faut alors retarder la décision sur le branchement, ou prendre une décision immédiate qui n'est pas forcément la bonne, sauf s'il est possible d'anticiper l'évaluation de la condition. Dans le contexte de vérification du flot de contrôle, certains cas d'aléas de contrôle peuvent conduire le dispositif de génération de signature à compacter des instructions qui ne seront pas exécutées par la suite, et bien évidemment ceci va poser des problèmes de cohérence pour la vérification.
Le cas particulier des prédictions de branchement sera traité plus en détail dans le paragraphe suivant ; nous allons nous attacher ici à la problématique générale et à une solution qui doit être la plus générale possible, vu que le comportement du processeur face aux aléas de contrôle est différent d'une architecture à une autre. Il existe quatre approches différentes de gestion des branchements : le « pipeline stall », l'anticipation des conditions de branchement, la prédiction ou encore le « branch delay slot ».

1. La première approche, appelée « pipeline stall », consiste à bloquer le pipeline le temps que l'instruction qui détermine le drapeau soit entièrement exécutée.

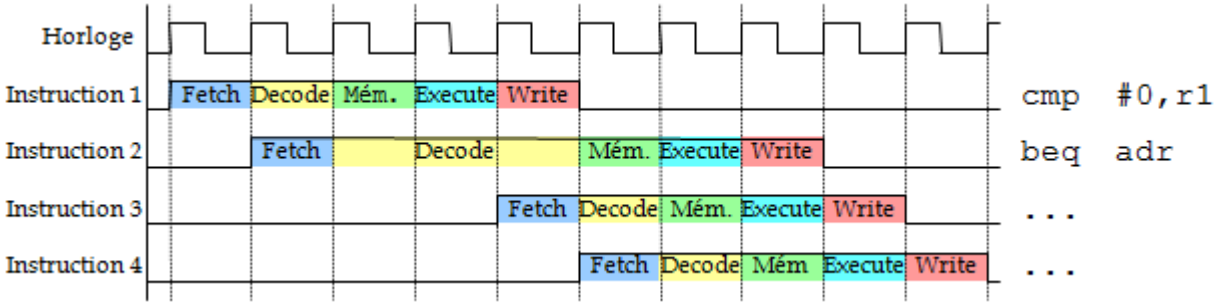


Figure 2-6 : Pipeline stall

⇨ Cette approche ne pose aucun problème pour la génération des signatures, puisque toutes les instructions chargées seront forcément exécutées.

Notons qu'intuitivement, le blocage du pipeline peut apparaître comme une solution assez intéressante pour empêcher les problèmes de cohérence des signatures. En effet, il suffirait de provoquer systématiquement un blocage pendant l'exécution des singularités pour que toutes les instructions qui vont être chargées soient exécutées.
Cette solution ne peut malheureusement pas être utilisée, car selon l'architecture du processeur vérifié, un blocage peut être imposé globalement à tous les étages du processeur et par conséquent ne servirait à rien. De plus, chaque blocage induit une diminution des performances.

2. La deuxième approche pour gérer les problèmes de branchement est fournie par le compilateur. Ce dernier peut dans certains cas anticiper l'instruction qui génère le drapeau ; le même genre d'approche est aussi utilisé de manière dynamique dans certains processeurs. Si le bon nombre d'instructions (défini par l'architecture) est intercalé entre la génération du drapeau et le saut, celui-ci peut s'effectuer sans délai. Il faut toutefois remarquer que cette anticipation est souvent impossible et qu'elle est très spécifique à une architecture donnée.

⇨ Comme avec un blocage, aucun problème ne se pose dans ce cas pour la génération des signatures car toutes les instructions chargées seront forcément exécutées.

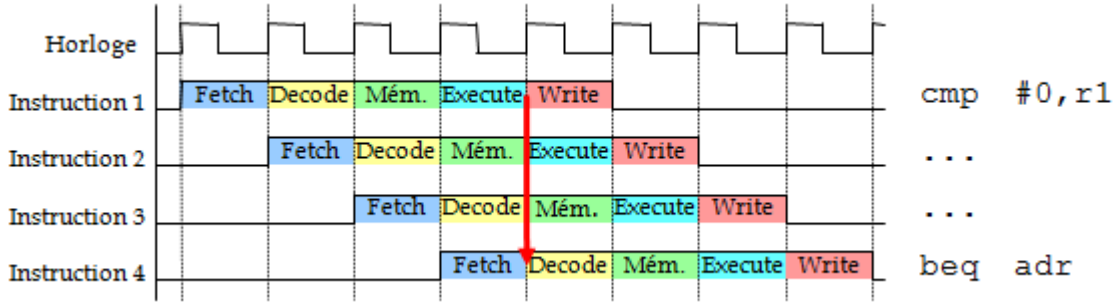


Figure 2-7 : Inversement de l'ordre des instructions par le compilateur

3. Une troisième approche est l'exécution spéculative après prédiction de branchement : le processeur peut essayer de deviner si le branchement sera pris ou pas et commencer à exécuter les instructions correspondant à cette décision. Si le choix se révèle correct, la pénalité de branchement est éliminée alors que si le choix se révèle incorrect, il faudra vider une partie du pipeline et charger l'instruction correcte. Ceci doit bien sûr être géré de manière cohérente au niveau de la génération de la signature à vérifier par le watchdog.

4. La dernière approche, appelée branchement retardé ou « branch delay slot », est utilisée dans de nombreux processeurs. Elle consiste à insérer, pendant la compilation, des instructions directement après le branchement.
   Selon la politique utilisée, le retardement peut être sans optimisation avec l'insertion de NOP, ou avec optimisation avec l'insertion d'instructions utiles, et dans ce cas, 3 techniques sont possibles :
   a. Déplacer une instruction précédente, fonctionnellement indépendante des autres instructions et du calcul de la condition de branchement.
   b. Dupliquer l'instruction de l'adresse de branchement.
   c. Laisser l'instruction suivante.

Face à cette panoplie d'approches de gestion des branchements, la solution proposée de manière à respecter au maximum la généralité de la méthode sera identique à la technique utilisée dans WDP. Le principe de cette solution consiste à ne traiter qu'un nœud à la fois :

- Pour les nœuds de destination, le fonctionnement du watchdog n'est pas modifié, ceux-ci étant toujours traités au moment de leur chargement.
- Pour les nœuds de séquencement, le traitement s'étale sur tout le temps pendant lequel l'instruction de séquencement est présente dans le pipeline du processeur.
  Pour cela, le watchdog calcule l'âge de l'instruction de séquencement (le nombre de lectures ayant eu lieu entre le début du chargement de cette instruction et l'exécution de celle-ci). En fonction de l'âge, le watchdog saura si le branchement a été effectué ou pas ou s'il y a eu un branchement inopiné. Cette solution implique que le watchdog traite les singularités non plus au moment de leur chargement par le processeur mais à retardement, donc en fonction de leur âge.

Cette solution nécessite également que le watchdog conserve toutes les signatures intermédiaires des instructions chargées par le processeur mais non encore exécutées.

Pipeline et interruptions :

La présence de pipeline complique le traitement des interruptions : lors du déclenchement d'une interruption non-masquable, la routine de traitement doit parfois être lancée immédiatement. Le pipeline contiendra alors des instructions partiellement exécutées.

La seule solution efficace est de vider partiellement le pipeline. Cette opération peut toutefois compliquer le redémarrage du programme après le traitement de l'interruption (par exemple, il faut retrouver l'adresse de la première instruction dont l'exécution n'a pas été achevée).

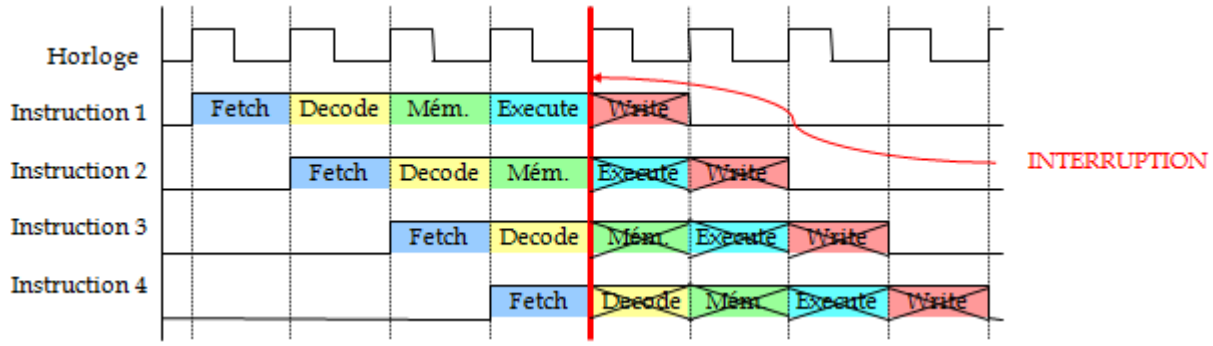


Figure 2-8 : Gestion des interruptions dans un pipeline

En cas de départ en exception sur une instruction de séquencement conditionnelle, le branchement peut avoir eu lieu sans que l'instruction destination ait été chargée. L'adresse du retour du processeur sera donc la destination du branchement et non pas celle située en séquence après la dernière unité chargée avant le départ.

Pour identifier ce cas gênant, lors du retour de l'exception, la solution la plus simple consiste à empiler l'âge du nœud en cours de traitement au moment du départ. De cette manière, le traitement du nœud interrompu peut reprendre normalement, en fonction de l'âge du nœud, lors du retour d'exception.

II.2.4. IDSM et la prédiction de branchement

La méthode présentée dans le paragraphe II.2.3 pour résoudre les problèmes que peuvent poser les aléas dans le pipeline est valable dans le cas des architectures qui adoptent la prédiction de branchement. Cependant cette méthode n'est pas la plus optimisée. C'est pour cette raison que nous nous sommes proposés de trouver une deuxième solution spécifique à ce type d'architecture. Mais avant de présenter cette solution, voici une brève présentation des différents types de prédiction :

⇨ Statique : la prédiction du branchement est fixe. Elle peut être définie en matériel au moment de la conception du processeur, ou bien être configurable. Il y a deux choix possibles pour la prédiction statique :

- Prédit pris (predict taken) : l'instruction chargée après un branchement est l'instruction destination lorsque le branchement est effectué. Cette technique est notamment utilisée dans le Leon3.
- Prédit non pris (predict not taken) : c'est l'inverse, et la stratégie la plus courante. La raison est d'ordre pratique : dans cette stratégie, les instructions exécutées spéculativement sont les instructions qui suivent immédiatement l'instruction de saut et qui sont vraisemblablement déjà stockées dans le cache.

⇨ Semi-statique : les sauts sont, en moyenne, pris une fois sur deux. Par contre, les sauts en arrière (boucles) sont plus souvent pris que pas pris, et inversement pour les sauts en avant. Cette observation est à l'origine d'une troisième stratégie de prédiction, qui tient compte de la direction.

⇨ Dynamique : la prédiction du branchement est définie au moment de l'exécution du programme, sur la base d'une analyse du comportement antérieur. Pour réaliser une prédiction dynamique, le processeur mémorise le comportement du programme lors de l'exécution des sauts. A chaque exécution d'un branchement dans le programme, le processeur mémorise si le saut était pris ou pas pris dans un tampon de prédiction de branchement (branch prediction buffer) ou une table historique de branchement (branch history table). Sur la base du comportement passé du programme pour un branchement donné, le processeur prédit son comportement pour l'exécution suivante du même saut.
Par rapport à la prédiction statique, la prédiction dynamique est plus performante et permet en plus de gagner du temps en ne recalculant pas à chaque fois les adresses de destination du saut, mais nécessite une quantité très importante de logique de contrôle. Cette approche est utilisée dans de très nombreux processeurs.

Pour garder la cohérence de la vérification du flot de contrôle, et notamment du calcul de signatures, face à l'exécution spéculative, le watchdog pourrait imiter le processeur en temps réel et prédire si le branchement va être pris ou non. Une première solution consiste à accéder à la table de prédiction de branchement BHT, une solution que nous voulons éviter puisque les signaux nécessaires pour le faire sont généralement des signaux internes au processeur, et nous voulons éviter au maximum de les utiliser.

Nous proposons donc de recréer dynamiquement la table de prédiction des branchements dans le watchdog et ce, par le biais de la surveillance des adresses utilisées par le processeur. Bien que ces signaux soient parfois aussi internes au processeur, leur utilisation est indispensable comme nous l'avons expliqué dans le paragraphe II.2.1.

Pour recréer la BHT nous pouvons procéder comme suit :

- Pendant les instructions de branchement conditionnel, le watchdog accède à sa table de prédiction pour savoir s'il doit pointer sur l'instruction de destination de son programme ou bien simplement incrémenter l'adresse de l'instruction lue. Le watchdog n'aura donc plus besoin d'attendre de voir si le processeur fait le saut ou non.
- Le cycle suivant, on vérifie si la prédiction était correcte ou pas, et on met à jour la table de prédiction du watchdog en fonction du résultat et ce, selon la machine à états illustrée dans la figure 2-9.

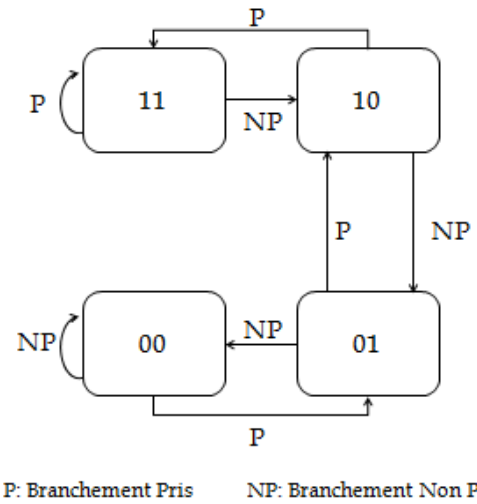


Figure 2-9 : Machine à états d'une entrée de BHT

Cette méthode améliore les performances du watchdog qui ne sera plus obligé d'attendre de voir si le processeur effectue le saut ou pas afin de le suivre. Néanmoins, elle risque d'augmenter considérablement la complexité du watchdog.

II.2.5. IDSM et le fenêtrage des registres

Dans un système sans fenêtrage, les registres deviennent très rapidement un goulot d'étranglement. En effet, chaque fonction ayant besoin de son propre espace de calcul, doit sauvegarder ses registres avant d'appeler une fonction et les restaurer au retour. Même si cette sauvegarde se fait sur le cache (accessible en quelques coups d'horloge), le temps perdu est considérable. Les systèmes à base de fenêtres de registres, comme les architectures Sparc, ont été développés pour palier à ce problème. Dans le Leon3, par exemple, les registres de calcul sont divisés en quatre groupes :

– Les registres globaux : ces registres sont accessibles de toute fenêtre à tout moment, ils sont donc partagés entre les fonctions.

– Les registres locaux : ces registres sont spécifiques à chaque fenêtre, donc à chaque fonction.

– Les registres d'entrée : ces registres contiennent les paramètres de la fonction, remplis par l'appelant, et éventuellement remplacés par le résultat de la fonction.

– Les registres de sortie : ces registres sont utilisés pour transmettre des paramètres à d'autres procédures et recevoir des résultats en retour.

Pour changer de fenêtre, il existe deux appels en assembleur :

– L'appel SAVE : cet appel est fait avant d'appeler une fonction ou au début de cette fonction, mais pas systématiquement. Pendant un appel SAVE les registres de sortie de la fenêtre courante deviennent les registres d'entrée de la nouvelle fenêtre et le processeur "alloue" de nouveaux registres locaux et de sortie.

– L'appel RESTORE : cet appel est fait une fois que la fonction est terminée, mais pas systématiquement. Pendant un appel RESTORE les registres d'entrée de la fenêtre courante deviennent les registres de sortie de la nouvelle fenêtre et le processeur "retrouve" les registres locaux et de sortie de la fonction vers laquelle on est retourné.

En quoi est ce que ce mécanisme est gênant pour la vérification de flot de contrôle ? Généralement, l'adresse de retour d'un sous programme est stockée dans un registre bien déterminé (registre %o7 pour les architectures Sparc v8). Or ce registre change de désignation avec le changement de fenêtre (par exemple : registre de sortie -> registre d'entrée). Dans le cas des appels imbriqués de fonction, il peut ne pas y avoir d'instruction SAVE entre chaque instruction CALL (on reste donc dans la même fenêtre). Quand le processeur désire revenir à la fonction du premier niveau (appelée fonction feuille), il rétablit sa fenêtre de registres en appelant la fonction RESTORE, et ce pour pouvoir accéder au registre stockant l'adresse de l'instruction appelante, en l'occurrence %o7 pour l'architecture Sparc. Le watchdog, de son côté, doit détecter le fait qu'on ne revient pas à la dernière fonction appelante mais à un niveau supérieur. Pour cela, il doit avoir un mécanisme pour imiter les instructions SAVE et RESTORE afin de savoir à quelle adresse lui aussi doit retourner. Ainsi, lorsque le watchdog n'exécute pas d'instruction SAVE entre deux instructions CALL, il sait que l'adresse de retour est celle de la fonction feuille, qui est la première fonction appelante.

II.3. Autres considérations : les routines d'initialisation et de terminaison et les fonctions des bibliothèques

D'une manière générale, l'application à vérifier avec la méthode IDSM est un programme exécutable compilé pour une architecture cible. Or tout exécutable, comme tout objet dynamique, contient, outre sa fonction spécifique, du code pour l'initialisation et la terminaison de l'exécution. Ces codes apparaissent après l'édition des liens et sont exécutés respectivement à chaque fois que l'objet dynamique est chargé dans l'environnement d'exécution ou déchargé de celui-ci.

D'autre part, l'exécutable à vérifier peut contenir des fonctions externes venant des bibliothèques qu'il utilise. En effet, presque tous les langages de haut niveau, fournissent des bibliothèques contenant des fonctions intégrées standard extrêmement utiles voire indispensables pour les développeurs comme "memcpy", "memcmp" ou encore "memmove". Ainsi, pendant la génération de l'exécutable et plus précisément pendant la phase de l'édition des liens, les bibliothèques sont liées d'une manière statique ou dynamique au programme à vérifier.

Au final, nous nous retrouvons donc avec un exécutable englobant plus que la simple traduction assembleur du code source, alors que le watchdog est tenu de contrôler toute l'exécution et pas seulement les fonctions spécifiques au programme à vérifier.

Pour toutes les instructions ajoutées pendant la phase de l'édition des liens, à savoir les fonctions des bibliothèques utilisées ainsi que les routines d'initialisation et de terminaison relatives au processeur, nous choisissons de faire une surveillance continue. En effet, ne disposant pas du code source de ces fonctions, et pour ne pas être amenés à parser leur code assembleur pour créer les GFCs correspondant et ainsi déterminer les singularités et les localiser, nous proposons d'associer une instruction watchdog à chaque instruction processeur, de telle manière que le watchdog puisse suivre le processeur instruction par instruction en calculant pour chacune d'elles une signature horizontale. Cette approche a évidemment un coût mémoire important et pourrait être optimisée, au prix toutefois de développements importants. Nous rappelons ici qu'aucune approche présentée dans la littérature ne prend explicitement en compte les fonctions appelées en dehors du source de l'application ou ajoutées lors de l'édition de liens. L'approche proposée permet aussi de surveiller l'exécution d'appels système.

II.4. Jeu d'instructions du watchdog

Le watchdog distingue les singularités suivantes : les instructions de branchement et de destination, les instructions fréquentes et celles qui correspondent à une opération mémoire sur une variable critique. De ce fait le jeu d'instructions de base du watchdog doit comprendre 14 instructions différentes, auxquelles peuvent s'ajouter des instructions spécifiques à une certaine architecture de processeur.

- I_0 : Destination
- I_1 : Destination load
- I_2 : Destination store
- I_3 : Instruction fréquente
- I_4 : Instruction load
- I_5 : Instruction store
- I_6 : Branchement conditionnel
- I_7 : Branchement inconditionnel
- I_8 : Branchement vers un sous programme
- I_9 : Retour d'un sous programme
- I_{10}* : Instruction de saut en dehors du programme principal
- I_{11}* : Instruction de non saut en dehors du programme principal
- I_{12}* : Branchement vers un sous programme en dehors du programme principal
- I_{13}* : Retour d'un sous programme en dehors du programme principal

*Instructions relatives aux fonctions générées pendant l'édition des liens

Une description détaillée du jeu d'instructions est présentée dans l'annexe –A- pour l'étude de cas sur le processeur Sparc v8 présentée dans le chapitre suivant. Deux instructions supplémentaires sont alors nécessaires pour gérer le fenêtrage de registres.

Revenons à l'exemple de calcul du PGCD présenté dans le chapitre I et utilisé pour décrire les méthodes CFC de l'état de l'art. Cet exemple simple, permet d'illustrer le principe de génération d'un programme pour le watchdog, pour la vérification d'une petite application sans appels extérieurs. Par contre, à cause de cette simplicité, seules les opérations de base sont utilisées, même en variant les seuils de fréquence et de criticité. Des exemples plus complets et plus fournis seront présentés dans le troisième chapitre.

La figure 2-10 montre le code assembleur obtenu suite à la compilation du code source de l'application PGCD pour l'architecture Sparc v8. Dans cet exemple, nous nous intéressons seulement au programme principal, les fonctions d'initialisation et de bibliothèques sont ignorées. Les figures 2-11 et 2-12 montrent respectivement le squelette et le GFC du programme watchdog correspondant.

```
main(){                          | 400011a0 <main>:
   int y = 12;                   | 400011a0:    84 10 20 0c mov  0xc, %g2
   int x = 5;                    | 400011a4:    82 10 20 05 mov  5, %g1
   int res ;                     | 400011a8:    80 a0 40 02 cmp  %g1, %g2
   while  (x != y) {             | 400011ac:    16 80 00 04 bge  400011bc
     if (x < y)                  | 400011b0:    01 00 00 00 nop
          y = y - x;             | 400011b4:    84 20 80 01 sub  %g2, %g1, %g2
     else                        | 400011b8:    30 80 00 02 b    400011c0
          x = x - y;             | 400011bc:    82 20 40 02 sub  %g1, %g2, %g1
   }                             | 400011c0:    80 a0 40 02 cmp  %g1, %g2
   res = x ;                     | 400011c4:    12 bf ff f9 bne  400011a8
                                 | 400011c8:    01 00 00 00 nop
}                                | 400011cc:    81 c3 e0 08 retl
                                 | 400011d0:    01 00 00 00 nop
```

Figure 2-10: Code source et code assembleur de l'exemple PGCD

Instructions processeur	Instructions watchdog correspondantes
P0: mov 0xc, %g2	**W0**: I0 –-Instruction de destination correspondant au début du main et à l'instruction **p0**
p1: mov 5, %g1	**W1**: I7 –-Saut inconditionnel vers **W2**
p2: cmp %g1, %g2	**W2**: I0 –-Instruction de destination correspondant à **p2**. Elle est accessible à partir de **W1** et de **W8**
p3: bge <**p7**> **p4**: nop	**W3**: I6 --Saut conditionnel vers **W6.** Selon le résultat de comparaison de %g1 et %g2, le processeur effectuera le saut ou pas. Le watchdog devra attendre le processeur et le suivre en cas de saut.
p5: sub %g2, %g1, %g2	**W4**: I0 –-Instruction de destination correspondant à **p5**
p6: b <**p8**>	**W5**: I7 --Saut inconditionnel vers **W7**
p7: sub %g1, %g2, %g1	**W6**: I7 --L'instruction de soustraction **p7** étant la seule instruction de son bloc (Voir figure 2-12), est considérée comme une instruction de saut inconditionnel vers **W7**.
p8: cmp %g1, %g2	**W7**: I0 –-Instruction de destination correspondant à **p8**
p9: bne <**p2**> **pA**: nop	**W8**: I6 --Saut conditionnel vers **W2.**Selon le résultat de comparaison de %g1 et %g2, le processeur effectuera une nouvelle itération de la boucle ou pas. Comme pour l'instruction **W3** le watchdog devra attendre le processeur et le suivre en cas de saut.
pB: retl **pC**: nop	**W9**: I9 --Fin de la fonction main

Figure 2-11: Squelette du programme du watchdog pour l'exemple PGCD

La figure 2-12 illustre le GFC correspondant au programme du watchdog pour le PGCD. Il est composé de 9 instructions regroupées en 6 blocs.

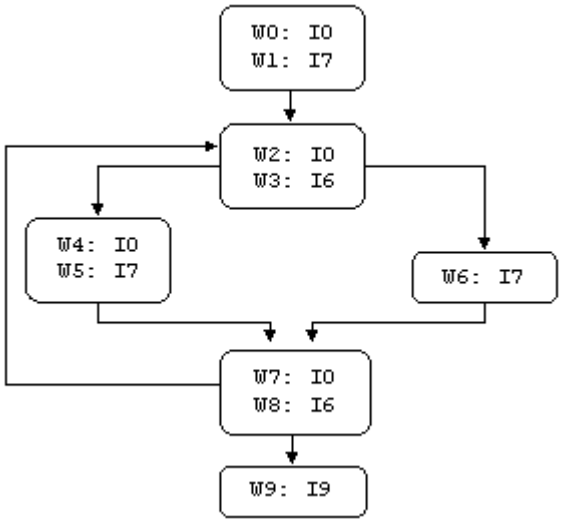


Figure 2-12: GFC du programme du watchdog pour l'exemple du PGCD

II.5. Architecture du watchdog

Pour satisfaire les contraintes de portabilité et d'adaptabilité, l'architecture interne de notre watchdog se doit d'être modulaire avec une séparation des différentes fonctions.

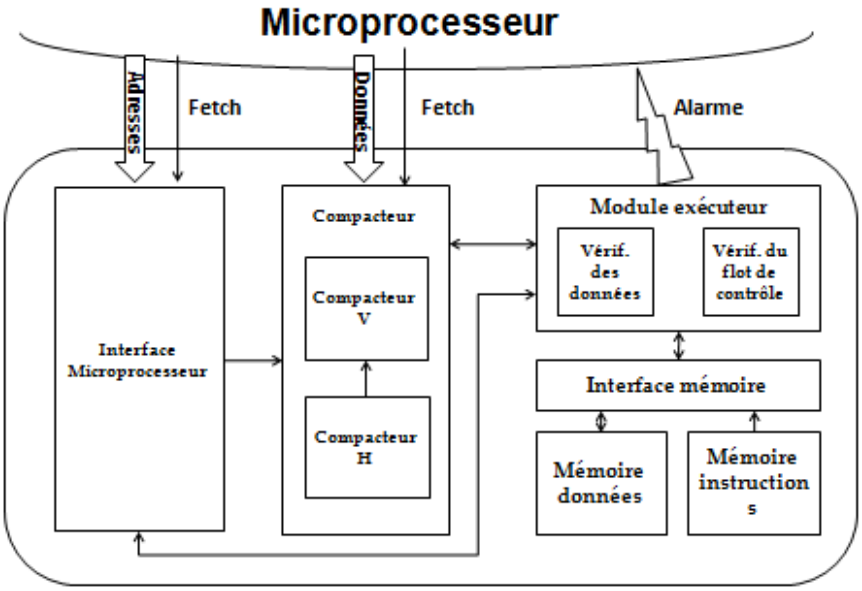


Figure 2-13 : Architecture interne du watchdog

L'architecture retenue, illustrée dans la figure 2-13, est une architecture à quatre modules communicants, qui vont être présentés de façon générique.

II.5.1. Module interface microprocesseur

Le module interface microprocesseur joue le rôle d'intermédiaire entre le système à surveiller et les autres modules du watchdog. Il reçoit en entrée le contenu du bus d'adresses du microprocesseur et certains signaux de contrôle. Il est chargé des tâches suivantes :

- Synchronisation du watchdog par rapport au programme du microprocesseur.
- Détection des ruptures de séquencement du microprocesseur.
- Signalisation de l'occurrence d'un nœud :
  - Il signale à la partie exécutrice l'occurrence d'un branchement.
  - Il reçoit les adresses relatives des singularités suivantes (du type instruction fréquente ou contrôle de variable critique) s'il y en a, et les fournit à la partie exécutrice.
- Ce module doit également calculer l'âge du nœud dans le cas d'un processeur pipeline.

II.5.2. Module compacteur

Ce module permet de générer les signatures en temps réel pour permettre à la partie exécutrice de les comparer avec les signatures stockées. Il peut être divisé en deux sous modules :

- Compacteur H : Il permet de générer les signatures horizontales sur 4 bits à partir d'une instruction sur 32 bits.
- Compacteur V : Il permet de générer une signature verticale sur 16 bits à la fin de chaque bloc du programme à partir des codes des instructions de ce bloc.

II.5.3. Module exécuteur

Ce module est chargé d'exécuter le jeu d'instructions du watchdog. Il ne dialogue pas directement avec le microprocesseur mais uniquement à travers les autres modules du watchdog. C'est lui qui dirige les autres modules en fonction des instructions à exécuter. Il peut être décomposé en deux sous modules :

- Module de vérification des données : en charge de vérifier l'intégrité des données et de mettre à jour les valeurs des variables.
- Module de vérification du flot de contrôle : en charge de suivre l'exécution des instructions sur le microprocesseur pour détecter les erreurs de flot de contrôle.

Un autre sous module est également à prévoir si nous choisissons de faire la prédiction des branchements pour les architectures à exécution spéculative (voir paragraphe II.2.4). Ce sous module sera en charge de faire la mise à jour de la table BHT et d'envoyer un signal de prédiction au module interface microprocesseur.

II.5.4. Module interface mémoire

Ce module est chargé de gérer les échanges avec la mémoire du watchdog, à savoir :

- Charger les instructions du watchdog
- Gérer les piles: ce module doit pouvoir empiler et dépiler les informations telles que le compteur de programme ou encore la signature intermédiaire.
- Gérer les opérations sur les variables critiques : il doit mettre à jour les valeurs des variables critiques ou aller chercher leur valeur lors des comparaisons.

II.6. Comportement général du watchdog

Le principe du comportement général du watchdog est assez simple. En effet, ce dernier attend que le processeur principal arrive sur une singularité pour exécuter son instruction correspondante. L'organigramme de la figure 2-14 résume le principe de fonctionnement du watchdog.

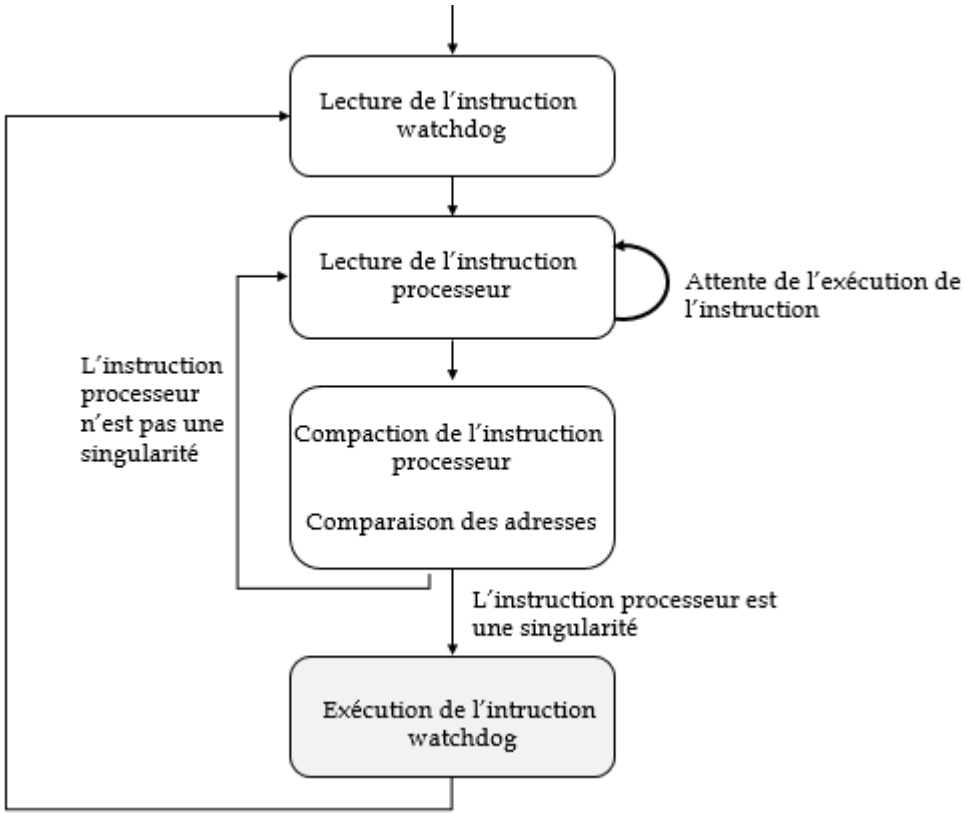


Figure 2-14 : Comportement général du watchdog

La quatrième étape de cet organigramme peut être décomposée en plusieurs sous états. Une description détaillée est illustrée dans la figure 2-15.

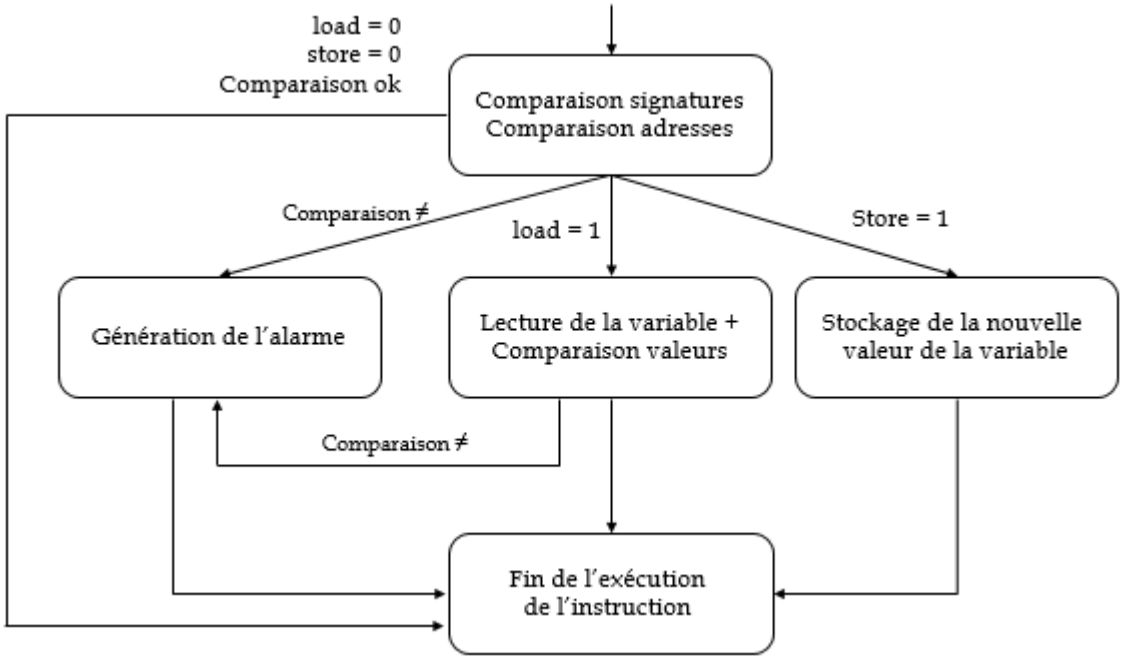


Figure 2-15 : Etapes de l'exécution d'une instruction watchdog

II.7. Discussion sur la couverture effective des erreurs

Dans le paragraphe II.1.4 nous avons présenté une étude théorique sur la couverture d'erreurs. Une étude plus précise doit être réalisée pour chaque implantation pratique de la méthode, compte tenu des contraintes d'implantation et des mécanismes inclus dans le processeur.

Par exemple, pour détecter toutes les erreurs sur les instructions de branchement, touchant l'ordre des instructions, à l'instant où elles se produisent, il faudrait comparer l'adresse de destination prise par le processeur avec une adresse complète dans l'instruction watchdog. Or ceci n'est pas possible si le processeur dispose d'une MMU. Pour un processeur doté d'un mécanisme de pagination et dont la taille de la page est de 4Ko (2^{12} bits), s'il effectue un saut vers une instruction de destination illégale dont les 12 premiers bits de l'adresse sont identiques aux 12 premiers bits de l'adresse de la destination légale, l'erreur n'est pas détectée. Cependant, un deuxième contrôle est effectué pour vérifier la légalité du saut, à savoir la comparaison de la signature horizontale. Le test effectué en début de bloc, et donc avec une signature intermédiaire nulle, revient à vérifier les 4 premiers bits de l'instruction et son efficacité dépend donc fortement de l'entropie du programme à tester.

Pour un processeur Sparc v8, par exemple, considérons un programme avec des instructions variées et une entropie élevée. On considère les trois événements suivants :

- A = « Non détection d'une erreur affectant l'ordre d'exécution des instructions par vérification de l'adresse »

  La probabilité de cet événement correspond à la probabilité qu'une erreur touche un bit non vérifié.
- B = « Non détection d'une erreur affectant l'ordre d'exécution des instructions par vérification de la signature horizontale »

  La probabilité de cet événement correspond à la probabilité que l'instruction de destination illégale commence exactement par les mêmes 4 bits que l'instruction de destination légale. Or, pour 4 bits vérifiés, il existe 2^4 combinaisons possibles dont une seule engendre un masquage de l'erreur.
- Non détection = A et B

La probabilité de détection d'une erreur affectant l'ordre d'exécution des instructions sur une instruction de branchement sans temps de latence est donc :

P (Non Détection) = P (A) x P (B) = 20/32 x $1/2^4$ = 3,9%

Avec les hypothèses citées, en particulier la variété des instructions du programme et son entropie, la probabilité de détection d'une erreur de flot de contrôle sur une instruction de branchement avoisine 96%. Cependant, si l'erreur n'est pas immédiatement détectée elle peut l'être à la fin du bloc avec la vérification de la signature verticale.

Par ailleurs, les erreurs sur le contenu des instructions peuvent ne pas être détectées dans certaines conditions. En effet, le mécanisme de compaction, qui va nous permettre de détecter les erreurs sur le contenu des instructions, a une probabilité de masquage intrinsèque. Pour la méthode de compaction sélectionnée, le taux de masquage tend vers 2^{-k}, k étant le nombre de

bits de la signature, lorsque la séquence à analyser est longue. Pour des séquences courtes (comme celles correspondant à un bloc linéaire typique), il n'existe pas de quantification a priori du taux de masquage. De plus, le nombre de bits des signatures a un impact direct sur le coût de l'implantation ; réduire ce coût revient donc à augmenter la probabilité de non détection.
Le choix de garder le même mécanisme de vérification des signatures en dehors du programme principal, pour ne pas augmenter la complexité du watchdog, présente aussi des inconvénients. En effet, le contrôle du contenu des instructions relatives au boot et aux fonctions de bibliothèque, se résume à la vérification des 4 premiers bits de chaque instruction. Tous les bits d'une instruction ne sont pas toujours utilisés, et parmi les 4 bits vérifiés, nous contrôlons les bits de format de l'instruction. La probabilité de masquage d'une erreur sur le contenu de l'instruction, en dehors du programme principal, peut toutefois atteindre 87,5% dans le pire cas.

Pour toutes ces raisons, une étude expérimentale du taux de couverture d'erreurs sera présentée dans le chapitre III. Cette étude est basée sur des injections de fautes réalisées sur un prototype de la méthode IDSM pour un processeur Sparc v8.

II.8. Conclusion

Dans ce chapitre nous avons décrit les grandes lignes de la méthode IDSM. C'est une méthode de type DSM avec plusieurs extensions flexibles et configurables par l'utilisateur. En effet, outre les contrôles classiques en début et fin de blocs, notre méthode permet de faire une surveillance continue de l'exécution pour les instructions fréquemment exécutées, ce qui permet de réduire le temps de latence.

La méthode IDSM permet non seulement de détecter tous les types d'erreurs de flot de contrôle possibles (sauf la correction des chemins pour des raisons liées aux surcoûts) mais aussi de vérifier l'intégrité des données critiques et de signaler leur corruption. En effet, la détection des erreurs de données est permise comme une extension de notre approche. L'identification des variables critiques a nécessité l'élaboration d'un nouvel algorithme qui prend en compte leurs durées de vies, les dépendances fonctionnelles et la participation dans les conditions de branchement.

Les idées présentées sont génériques et peuvent être appliquées à un grand nombre de systèmes embarqués. Ces principes génériques doivent ensuite être déclinés en fonction des caractéristiques du processeur à surveiller et des coûts acceptables pour l'application. L'implantation reste malheureusement très liée à l'architecture du processeur visé, étant donné que les processeurs peuvent présenter plusieurs mécanimes gênants pour la vérification de flot de contrôle. Des solutions ont été trouvées pour contourner les effets de ces mécanismes dans certains cas, et la méthode proposée peut être implantée pour un microprocesseur avec mémoire cache, pipeline et MMU. Certains mécanismes conduisent toutefois à diminuer la probabilité de détection d'erreur. Les compromis à réaliser en fonction des diverses contraintes de coûts (mémoire, performances, complexité matérielle du watchdog, consommation) conduisent aussi à diminuer le pourcentage de détection d'erreurs. Un prototype utilisant la

méthode proposée sera présenté dans le chapitre 3 et ces aspects seront plus précisément discutés.

Outre les caractéristiques du processeur surveillé, d'autres contraintes ont été prises en compte, comme la synchronisation du watchdog et du processeur dans la phase de démarrage (avec la considération des routines d'initialisation et de terminaison), ainsi que la surveillance des fonctions des bibliothèques (ou fonctions système). Les choix proposés vont dans le sens de coûts élevés au niveau de la mémoire du watchdog, pour augmenter la probabilité de détection sur ces phases d'exécution, qui n'ont jusque là pas été prises en compte par les méthodes publiées alors qu'elles constituent une part significative du temps d'exécution. D'autres choix pourraient être faits, pour réduire les coûts en limitant la vérification, comme dans les approches précédentes, aux fonctions spécifiques de l'application.

Enfin, une ébauche de l'architecture du watchdog a été présentée. Cette architecture est modulaire pour pouvoir l'adapter aux contraintes de plusieurs architectures à vérifier avec le minimum de modifications. Une implantation va être décrite dans le chapitre suivant, pour un processeur d'architecture Sparc v8 : le Leon3.

Chapitre III :

Développement d'un prototype pour le processeur Leon3

Pour tester la méthode IDSM et évaluer ses différents coûts, nous avons développé un prototype utilisant notre méthode de vérification de flot de contrôle. Ce prototype a été soumis par la suite à des injections de fautes pour déterminer l'efficacité de la méthode ainsi que son taux de couverture réel, notamment pour les erreurs multiples.

Nous décrirons dans un premier paragraphe l'environnement adopté, puis nous nous intéresserons dans les paragraphes III.2 et III.3 aux outils logiciels développés ainsi qu'aux blocs du watchdog implantés. Le paragraphe III.5 quant à lui sera réservé à la description de la méthode d'injection de fautes choisie pour valider notre méthode. Mais avant d'évaluer l'efficacité de la méthode IDSM et ses différents coûts, dans les paragraphes III.6 et III.7 respectivement, la méthode de calcul de criticité sera également étudiée sur différentes architectures dans le paragraphe III.4.

III.1. Présentation générale du prototype

Maintenant que les principes généraux de notre méthode de vérification de flot de contrôle sont définis, nous nous proposons ici de l'adapter aux caractéristiques intrinsèques d'un processeur d'architecture Sparc v8.

Notre choix s'est posé sur le processeur Leon3, un processeur RISC de 32 bits, disponible comme un softcore, open source (sauf la version tolérante aux fautes), et téléchargeable sur le site de Gaisler Research. Il a été développé pour des applications embarquées (notamment pour le spatial) avec les caractéristiques suivantes : pipeline de 7 étages avec une architecture Harvard donc cache instructions et cache de données séparées, multiplieur et diviseur matériel, unité de debug et des extensions multiprocesseur. L'utilisation de ce processeur est en principe basée sur les interconnexions au travers d'un bus AMBA. Ce bus permet de faire un réseau de processeurs Leon3 ou d'ajouter de nombreux périphériques. La configuration que nous utilisons est une configuration de base. En effet, nous n'utilisons pas d'unité de calcul sur les nombres à virgules flottantes, l'unité principale étant l'unité de calcul sur les nombres entiers.

Avant de s'intéresser au Leon3, un prototype initial a été réalisé, basé sur le processeur Leon2, mais pour des raisons d'obsolescence de ce dernier nous avons dû adapter le travail réalisé au processeur Leon3 qui est un peu plus complexe. La version finale du watchdog réalisée est donc configurable, permettant de choisir auquel de ces deux processeurs elle sera connectée. Les processeurs Leon2 et Leon3 sont tous les deux des modèles VHDL open source d'un cœur de traitement 32 bits, entièrement conformes à la norme IEEE-1754 relative aux spécifications Sparc v8. Cependant, il existe des différences notables entre ces deux processeurs. Par exemple,

le Leon3 supporte le Multi-processing symétrique (SMP) et possède un pipeline sur sept étages, tandis que Leon2 ne supporte pas le SMP et dispose d'un pipeline sur cinq étages seulement.

Le watchdog est greffé au Leon3 de façon à ce qu'il puisse surveiller le flux circulant entre la CPU et la mémoire cache. Le watchdog renvoie également un signal d'erreur au processeur en cas de détection d'erreur. Dans ce prototype, le volet de vérification des données critiques ne sera pas considéré dans le watchdog matériel. Compte tenu des difficultés rencontrées et du temps disponible, seules les erreurs de flots de contrôle ont été traitées. Toutefois, tous les outils logiciels liés à l'évaluation de la criticité des variables ont été développés et nous en montrerons l'utilisation pour plusieurs processeurs, avant de les réutiliser dans l'étude du chapitre suivant.

Comme nous l'avons vu dans le paragraphe II.2., plusieurs ajustements peuvent être apportés à l'architecture de base du watchdog pour contourner les particularités de chaque processeur pouvant gêner la vérification de flot de contrôle. Dans le cas du Leon3, toutes les solutions proposées dans le paragraphe II.2 pour prendre en considération la mémoire cache, la MMU et le pipeline, peuvent être adoptées. Par ailleurs, pour gérer les mécanismes de fenêtrage de registres, deux instructions supplémentaires sont ajoutées afin de pouvoir mimer le Leon pendant ses instructions SAVE et RESTORE. Ces caractéristiques particulières ont eu un impact notable, à la fois sur le temps de développement du watchdog et sur sa complexité matérielle.

Pour étudier le comportement de notre système, Leon 3 + watchdog, nous avons choisi l'algorithme de chiffrement/déchiffrement AES pour s'exécuter sur le Leon 3. Cette application est complète car elle comporte toutes les singularités possibles dont notamment les appels de fonctions. Elle est de plus de taille moyenne mais notre prototype devrait se comporter de la même manière au regard d'une application plus grande.
L'objectif est de démontrer le niveau de détection atteint et d'identifier des cas où l'approche doit être complétée ; certains chiffres fournis peuvent être dépendants de l'application, mais certains d'entre eux n'ont pas de raison d'être différents, comme le niveau de détection atteint pour les erreurs dans le compteur de programme ou dans le registre d'instructions.

Une fois notre watchdog adapté et greffé au Leon, ce système (Leon+watchdog) a été implanté sur un FPGA Xilinx de la famille Virtex V.

III.2. Développement des outils pour la génération du programme du watchdog

Pour garantir la portabilité de notre méthode de vérification de flot de contrôle, il faut que le programme du watchdog soit généré indépendamment du langage source de l'application à vérifier. Il est également souhaitable que notre méthode puisse être utilisée pour différentes architectures cibles, avec le minimum de modifications. En effet, bien que les adaptations de la méthode soient inévitables pour prendre en considération les caractéristiques du processeur vérifié, comme nous l'avons vu dans le paragraphe II.2, il faut que, dans l'ensemble, les outils soient ré-utilisables d'une architecture à une autre, et en particulier les outils de génération du

programme du watchdog, qui doivent être si possible génériques et indépendants des particularités du processeur et de son jeu d'instructions.

Cette génération peut être faite (ou du moins commencée) pendant la phase de compilation de l'application : comme illustré par la figure 3-1, le compilateur permet à la fois de générer le fichier exécutable pour le processeur et le programme de vérification pour le watchdog. De cette façon, tout le mécanisme de génération du programme du watchdog est transparent pour le programmeur. L'utilisation du compilateur modifié nous permettra également de bénéficier des outils de calcul de durée de vie de variables, nécessaires pour le calcul de criticité de notre méthode. En effet, comme expliqué dans l'Annexe –B-, nous ne proposons pas de nouvelles techniques pour calculer la durée de vie, mais nous profitons des analyses précises existant dans les compilateurs modernes. Le calcul des durées de vie des variables, par exemple, est effectué sur l'ensemble du CDFG en utilisant des macros GCC dédiées et utilisées nativement pour faciliter l'allocation des registres.

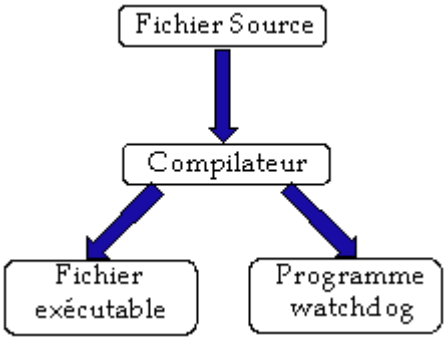


Figure 3-1 : Mécanisme initial de génération du programme du watchdog

Cependant, la génération du programme du watchdog au cours de la phase de compilation (et donc avant l'édition des liens) ne permet de tenir compte ni des appels à des fonctions de bibliothèques ni des appels système. Ainsi, tout le code inséré dans le programme exécutable final, après l'étape d'édition des liens, ne peut pas être vérifié.

Par ailleurs, étant donné que la phase d'édition des liens insère des bouts de code dans le programme, les adresses obtenues pendant la phase de compilation sont toutes relatives au début du main. En d'autres termes, si le programme inclut des appels système et des appels à des bibliothèques, les adresses calculées pendant la compilation ne sont pas les adresses utilisées au cours de l'exécution du programme. La conséquence directe de ces différences dans les adresses est la génération de signatures erronées.
Par conséquent, la génération du programme du watchdog doit être décomposée en deux étapes, comme illustré dans la figure 3-2, avec une phase en pré-édition des liens et une seconde phase en post-édition des liens :

- Pré-édition des liens : extraire le squelette du programme principal
- Post-édition des liens : essentiellement pour (1) prendre en considération les instructions ajoutées après édition des liens et (2) calculer les signatures.

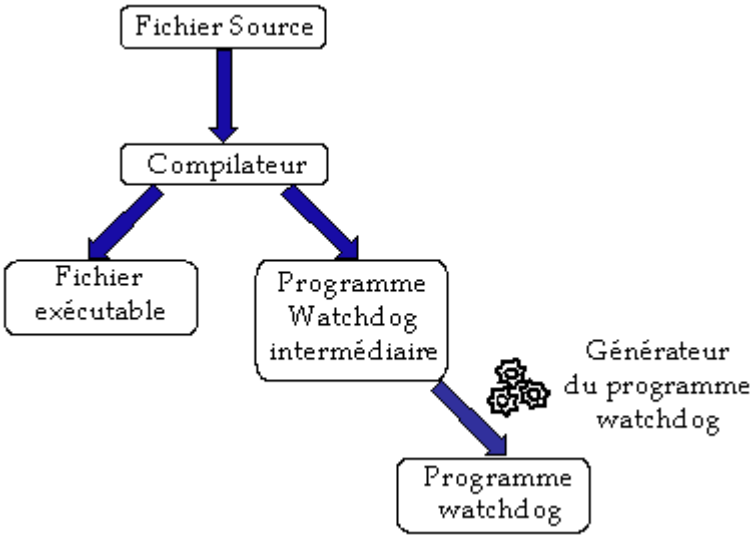


Figure 3-2: Chaîne complète de génération du programme du watchdog

La disponibilité de la suite de développement logiciel standard GNU-Linux, dont notamment GCC, est une des raisons qui a amené au choix du compilateur GCC. Les autres raisons sont surtout liées à la licence et à la portabilité. En effet, la suite GCC (GNU Compiler Collection) est un ensemble de composants conçus pour fournir à l'utilisateur la possibilité de compiler une grande variété de langages sources (C, C++, Objective-C, Fortran, Java, et Ada), vers une grande variété de processeurs cibles. Ce compilateur est libre d'utilisation.
Pour les premiers travaux, nous avons utilisé la chaine de cross-compilation GCC dans sa version 4.2.4. Les sources du compilateur ont été modifiées pour générer le programme du watchdog. Cette chaine de cross-compilation n'offrait que deux options pour l'édition des liens : (1) link statique qui implique l'insertion de toutes les fonctions, et pas seulement celles utilisées, venant des bibliothèques appelées dans l'exécutable ce qui augmente considérablement la taille de ce dernier et (2) link dynamique qui délègue à l'OS les tâches de relogeage et d'exécution des fonctions des bibliothèques précompilées et qui conduit à un fichier exécutable incomplet. Nous avons dû alors migrer vers une version modifiée du cross-compilateur GCC V4.4.2 pour Sparc v8, dont les sources ont été téléchargées à partir du site de Gaisler Research. Dans cette chaîne de cross-compilation, une troisième option d'édition des liens est possible. En effet, cette version permet d'inclure dans l'exécutable toutes les fonctions des bibliothèques utilisées sans pour autant inclure les bibliothèques dans leur totalité.

III.3. Implantation des blocs du watchdog

III.3.1. L'interface microprocesseur

Le but de l'unité d'interface avec le microprocesseur est de mettre les informations disponibles dans un état défini pour les autres unités quelque soit l'état du processeur principal. Cette description est paramétrée avec les informations décrivant le processeur principal.

Selon le principe exposé au chapitre II, cette interface devait recevoir en entrée le contenu du bus d'adresses du microprocesseur, les bus de données et d'instructions et certains signaux de contrôle. Pour notre prototype, ces informations sont extraites juste après leur lecture dans le pipeline, c'est-à-dire en entrée de l'étage de décodage, plutôt que dans l'étage de lecture d'instruction. Certains signaux de contrôle devant être extraits à ce niveau là, il était plus simple de ne modifier que cette partie du modèle du Leon3 pour l'extraction de l'ensemble des informations utiles. Nous reviendrons sur les conséquences lorsque nous discuterons les résultats des injections de fautes.

Cette interface est structurée en deux parties. La première permet de déterminer l'état du système afin de synchroniser le processeur et le coprocesseur. La deuxième permet de mimer partiellement la structure interne du processeur pour pouvoir donner à l'unité d'exécution du watchdog des données structurées indépendamment du processeur.
Le bloc de synchronisation permet de vérifier la plage de mémoire qui est utilisée. En effet le processeur Leon3 exécute les programmes que nous lui avons fournis et situés dans une zone spécifique de la mémoire. Par défaut cette plage commence à x''40000000'', mais c'est une donnée configurable si nécessaire.
Le bloc de synchronisation utilise différentes informations provenant du processeur. Pour connaître l'état du Leon nous avons un signal très important appelé « holdMicro ». Ce signal permet de déterminer si le pipeline du Leon est actif, si c'est le cas alors cela signifie que les informations prises sur le bus d'instructions doivent être prise en compte. Un deuxième signal est très important, c'est celui que nous avons appelé « annul_bit ». Ce signal apparaît dans l'étage des exceptions du Leon. S'il est levé alors l'instruction étant dans le même étage n'est pas exécutée. C'est pourquoi il faut attendre cet étage pour savoir s'il faut prendre en compte cette instruction.
Le deuxième bloc de l'unité est en fait une sorte de faux pipeline, illustrée par la figure 3-3. Ce deuxième bloc crée également les deux signaux d'horloges, en opposition de phase, utilisés pour synchroniser l'ensemble.

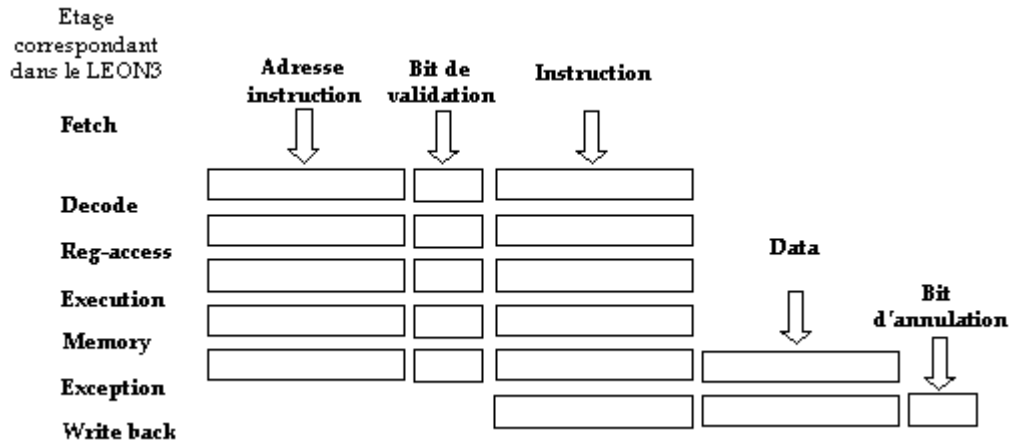


Figure 3-3: Les registres à décalage de l'interface microprocesseur jouant le rôle de faux pipeline

III.3.2. L'unité de compaction

L'unité de compaction est constituée d'un MISR permettant de faire une division polynomiale récursive sur 32 bits en utilisant ce qui vient du bus d'instruction du processeur principal, les bits de poids faible servant de signature. Cette unité contient également une pile permettant de sauvegarder des signatures intermédiaires avant de faire certains sauts (Voir figure 3-4).

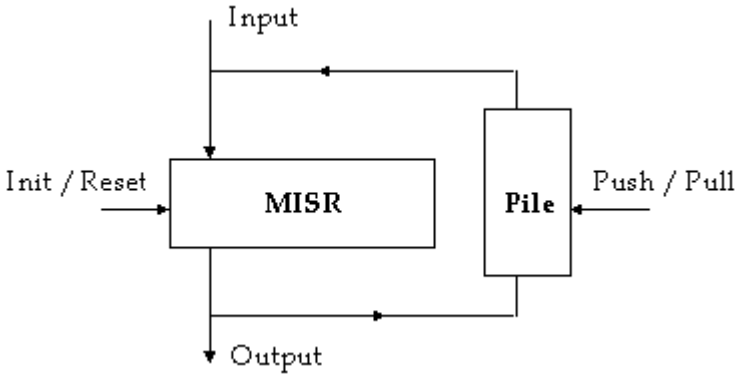


Figure 3-4: Schéma simplifié de l'unité de compaction

III.3.3. L'unité d'exécution

L'unité d'exécution est de loin la plus complexe du watchdog et aussi la plus importante en nombre de composants (autour de 60% du watchdog). C'est cette unité qui dirige l'exécution des instructions, effectue les vérifications et commande l'interface mémoire et l'unité de compaction. De plus, cette unité reçoit de nombreuses informations provenant à la fois de l'interface microprocesseur, de l'interface mémoire et de l'unité de compaction.

Les différents rôles de cette unité doivent être synchronisés avec les autres blocs du watchdog. Pour certaines instructions, nous avons même eu besoin que plusieurs blocs synchrones soient utilisés successivement dans le même cycle pour éviter de ralentir le processeur. Il faut alors utiliser une synchronisation à la fois sur front montant et sur front descendant.

La figure 3-5 permet d'illustrer le flux de données dans l'architecture au cours du temps. L'unité d'exécution a été éclatée pour avoir une vision plus précise des dépendances. Le décodage et les vérifications sont purement combinatoires. Le schéma représente dans quelle partie du cycle leur calculs sont significatifs.

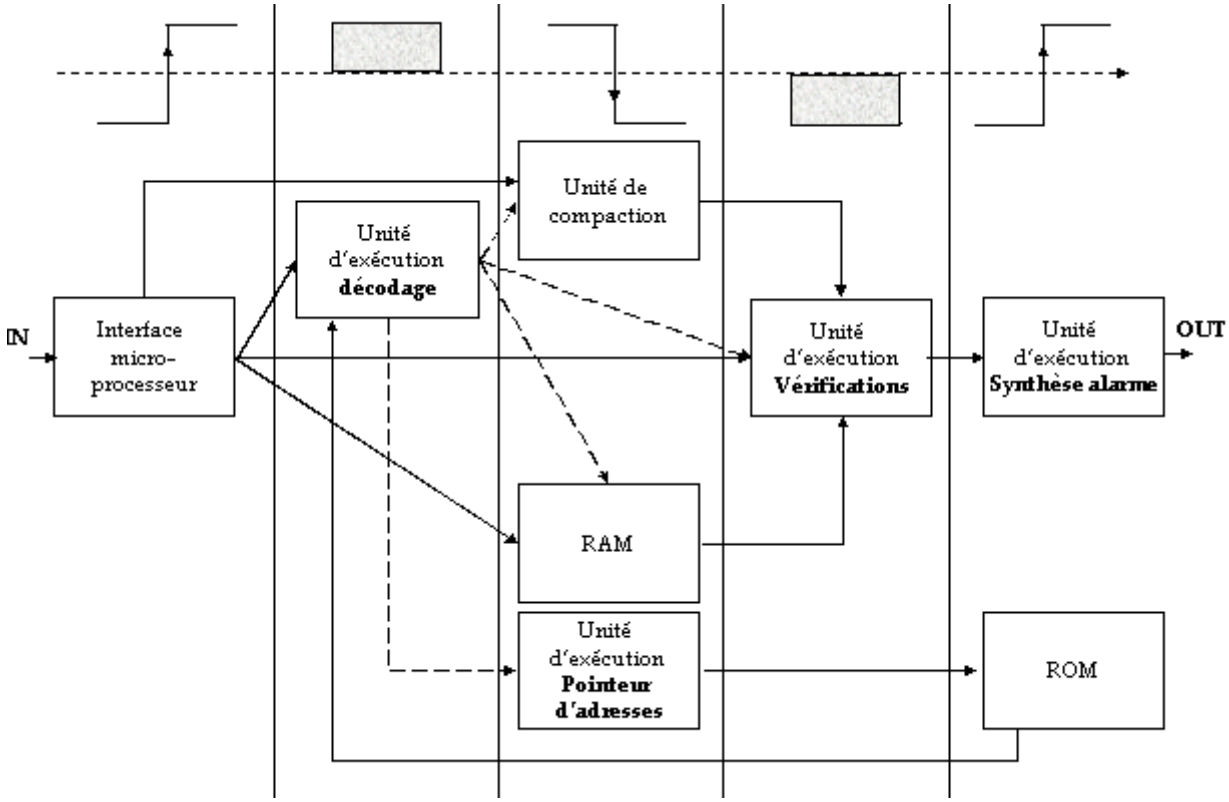


Figure 3-5: Flux de données dans le watchdog

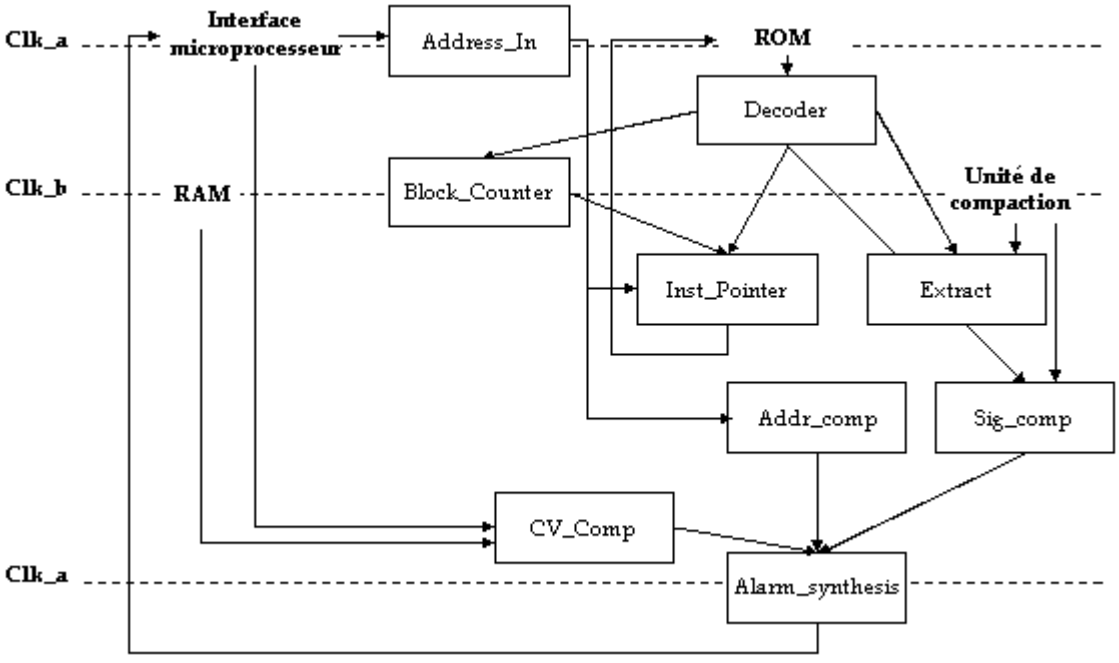


Figure 3-6: Schéma de l'unité d'exécution

Comme illustré dans la figure 3-6, l'unité d'exécution est découpée en 9 blocs :

- **Address_in :** Ce bloc utilise les adresses provenant de l'interface microprocesseur pour détecter si les adresses se suivent ou non, émettant ainsi un signal « not_adjacent_addr » réceptionné par le bloc instruction_pointer.
  Ce bloc délivre aussi en sortie l'adresse précédemment reçue.

- **Decoder :** Ce bloc permet de séparer les différents champs contenus dans l'instruction du watchdog. Les détails des champs sont présentés dans l'annexe A.
  Ce bloc envoie aussi les signaux de contrôle à différents blocs de l'unité, notamment les blocs de comparaisons, ainsi qu'à l'unité de compaction.

- **Extract :** Certains champs du jeu d'instruction comportent des informations hachées pour diminuer l'espace utilisé dans l'instruction. Les détails de ces champs sont présentés dans l'annexe A. Dans plusieurs formats d'instructions, les adresses de sauts, permettant au watchdog de suivre le Leon, ou encore les adresses de destination, permettant de contrôler l'arrivée du Leon à bon port, sont hachées avec les signatures. Ce bloc permet donc de décoder les champs hachés en utilisant les signatures calculées en ligne, pour récupérer les adresses.

- **Inst_pointer** : Ce bloc permet simplement d'incrémenter l'adresse de l'instruction lue. Il permet également de déterminer si un saut est possible. En effet, si le saut n'est pas autorisé et que les adresses n'étaient pas adjacentes alors ce bloc envoi un signal d'erreur au bloc alarm_synthesis.

- **block_counter :** Connaissant la taille d'un bloc, à partir des informations fournies par le décodeur, ce bloc permet de faire le décompte des instructions de ce bloc à chaque front montant de l'horloge clk_b. Ce mécanisme permet de déclencher l'instruction de fin de bloc du watchdog quand le Leon arrive sur une instruction de saut.
  Ce bloc permet également de faire une comparaison entre les adresses du Leon et les adresses relatives de référence marquant le moment où certaines instructions doivent être effectuées (lecture et écriture des variables critiques).

- **cv_comp, sign_comp, addr_comp :** Ces trois blocs sont très semblables entre eux. Ils permettent de comparer deux valeurs à leurs entrées et d'envoyer un signal d'erreur correspondant au bloc alarm_synthesis au cas où ces valeurs ne sont pas égales.

- **alarm_synthesis :** Ce bloc reçoit tous les signaux d'erreur des différents blocs et en fait la synthèse. Quand tous les signaux sont dit valides, le signal d'alarme est enregistré et envoyé à l'interface microprocesseur, au front montant de l'horloge clk_a.

Ces blocs peuvent être regroupés en fonction du moment du cycle d'horloge où ils interviennent. Les traitements à effectuer étant assez simples nous pouvons les séparer en deux parties, la première partie étant essentiellement du décodage et la deuxième étant principalement les comparaisons :

- Au front montant de l'horloge principale (clk_a) une instruction est lue et l'interface microprocesseur envoie l'adresse lue sur le bus du processeur principal.
  Le décodage se fait, les signaux de contrôle sont positionnés.

- Au front descendant suivant (clk_b) la RAM, le compteur et le compacteur sont déclenchés.
  Le pointeur d'instruction calcule l'adresse de l'instruction suivante, les comparaisons sont faites. Au front montant suivant (clk_a en bas) le dernier bloc fait une synthèse globale des signaux d'erreur reçu et renvoie le signal d'alarme à l'interface microprocesseur.

III.3.4. L'interface mémoire

Le rôle de l'unité d'interface mémoire est de gérer les accès et les possibilités de modifications de la mémoire du watchdog, en ajoutant par exemple un niveau de cache, mais aussi de gérer les spécificités éventuelles du stockage en mémoire.

Par ailleurs, une amélioration de cette interface peut être apportée concernant la gestion de réalignement des instructions lues sur le bus d'instructions. En effet, il est vrai que pour notre prototype, nous avons opté pour le schéma « une instruction par ligne », (Voir paragraphe A.4) mais une optimisation de cette organisation peut être envisagée pour des travaux ultérieurs étant donné que les instructions sont de tailles variables. Or, une organisation différente de la mémoire peut engendrer une différence d'alignement entre ce qui est lu dans la mémoire physique et ce qui est attendu sur le bus d'instruction, et cette différence doit pouvoir être gérée par l'interface mémoire.

III.4. Validation de la méthode de calcul de criticité

III.4.1. Méthodologie de validation

Afin de valider notre méthode de calcul de criticité et d'identification des registres critiques, on se propose ici d'évaluer la criticité des registres pour un ensemble de benchmarks, et d'effectuer par la suite une analyse empirique de sûreté sur le banc de registres du processeur. Cette analyse de sûreté servant de référence de comparaison peut être menée grâce à des techniques d'injection de fautes. Le principe est de comparer le comportement nominal du circuit (sans injection de fautes) avec son comportement en présence de fautes, injectées lors de l'exécution d'une application. Les applications choisies comme benchmark pour notre étude, et qui seront compilées avec notre version modifiée de GCC puis exécutées sur le processeur cible sont FIR, un filtre à réponse impulsionnelle finie, Mtmx, une application de multiplication de matrices, et Sieve, une application de calcul des nombres premiers utilisant le crible d'Ératosthène. Rappelons que la version modifiée du compilateur GCC permet de générer des informations complémentaires sur le flot de données à savoir la criticité des variables, leurs durées de vie ... Ces informations sont produites dans des fichiers textes facilement manipulables pour effectuer nos différents calculs et statistiques.

III.4.2. Validation de la méthode sur une architecture simple

On se propose dans un premier temps de valider notre méthode sur une architecture simple, le MC 68hc11. Ce microcontrôleur 8 bits possède 8 registres internes, traite des données 8 bits, mais peut toutefois travailler sur des données 16 bits par le biais d'une concaténation des deux accumulateurs A et B. La figure 3-7 illustre l'ensemble de ces registres.

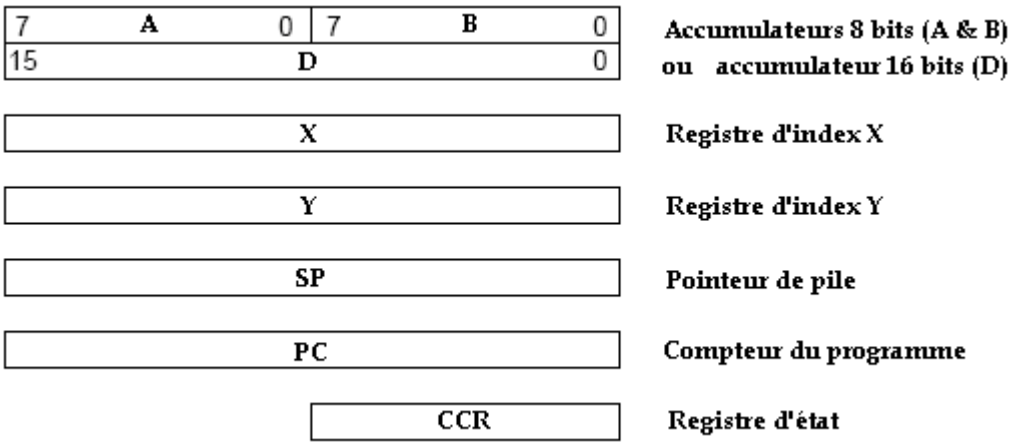


Figure 3-7 : Registres internes du 68hc11.

Lors de la compilation, GCC utilise, outre ces 8 registres internes, des registres intermédiaires ou temporaires pour la gestion des données (l'adressage des zones mémoires, l'empilement…). Parmi ces registres, nous pouvons citer le FP (frame pointer), d1 et d2 (des accumulateurs temporaires)... Le registre PC est, par contre, absent lors de l'analyse effectuée par GCC.
Pour cet exemple simple, des campagnes d'injection de fautes ont été réalisées à base de simulations, à l'aide de scripts TCL (Tool Command Language). Ces scripts ont permis de faire les simulations des fautes injectées l'une après l'autre, en retournant comme résultats l'effet de chaque injection sur le comportement du processeur. Le choix des cibles et des cycles d'injections a été fait d'une manière aléatoire. Le nombre de fautes injectées a été calculé selon le principe qui sera présenté plus en détail dans le paragraphe III.5.2, pour une marge d'erreur de 5% et un taux de confiance de 95%.

Le tableau III-I et la figure 3-8 montrent que les résultats de calcul de criticité permettent de donner une idée générale de robustesse des registres. Cependant, la prise en considération des registres temporaires pendant l'analyse théorique de criticité engendre une différence légère entre les résultats obtenus par les simulations et par l'approche théorique.
Néanmoins, si les registres d'index sont considérés comme une seule entité (comme nous le faisons pour les registres accumulateurs A et B) au lieu de regarder les criticités de X et Y indépendamment, nous retrouvons des classements de criticité assez proches pour les deux approches, comme résumé dans le tableau III-II. Par ailleurs, si nous faisons une correspondance entre les registres ''softs'' de GCC et les registres du 68hc11, en affectant les résultats de criticité de ces registres ''softs'' aux registres du 68hc11 qu'ils représentent au cours de la compilation, nous obtenons des résultats plus précis pour chaque registre.

Tableau III-I : Evaluations de criticité des registres du 68HC11 pour les applications Fir, Mtmx et Sieve

	Registres	Durée de vie	Fanout	Dépendances	Participation au saut	Criticité
FIR	X	6	3	566	9	48,90%
	D	54	4	614	24	100,00%
	Y	0	2	370	16	44,63%
	SP	0	0	0	0	0,00%
	FP	64	2	0	22	67,40%
	d1	9	1	408	1	29,78%
	d2	0	1	408	1	24,83%
	d3	0	0	0	0	0,00%
	d4	0	0	0	0	0,00%
MTMX	X	2	1	509	2	41,29%
	D	46	5	499	28	100,00%
	Y	0	2	188	23	44,65%
	SP	0	0	0	0	0,00%
	FP	67	2	0	27	73,65%
	d1	12	0	181	0	20,05%
	d2	0	1	181	3	17,35%
	d3	12	0	181	0	20,05%
	d4	0	1	181	3	17,35%
SIEVE	X	39	40	1109	16	66,33%
	D	61	42	1314	32	100,00%
	Y	0	1	1278	2	37,97%
	SP	84	40	772	9	68,55%
	FP	78	2	0	8	43,23%
	d1	15	2	363	6	23,56%
	d2	0	2	363	6	17,01%
	d3	0	0	0	0	0,00%
	d4	0	0	0	0	0,00%

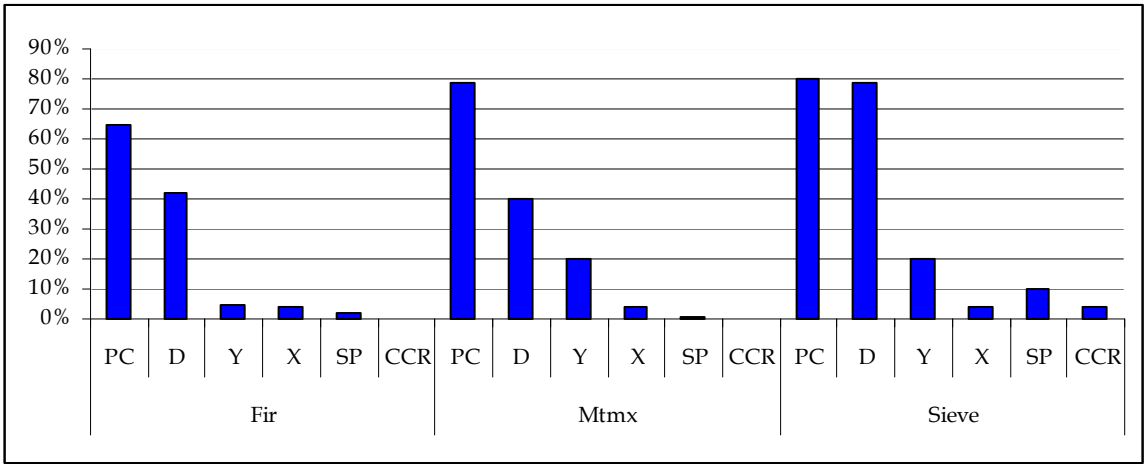


Figure 3-8: Résultats d'injections de fautes sur les registres du 68HC11 pour les applications Fir, Mtmx et Sieve : pourcentage des erreurs perturbant l'application

Tableau III-II: Classement des registres (hors PC) par criticité décroissante selon les deux approches

	Classement criticité	Classement injections		Classement criticité	Classement injections		Classement criticité	Classement injections
FIR	D	D	MTMX	D	D	SIEVE	X+Y	D
	X+Y	X+Y		X+Y	X+Y		D	X+Y
	Soft	SP		Soft	SP		Soft	SP
	SP	autres		SP	autres		SP	autres

III.4.3. Validation de la méthode pour le prototype Leon3

III.4.3.1. Contraintes liées aux registres du pipeline

Après l'étude sur le 68HC11, on se propose dans ce paragraphe d'évaluer la criticité des registres pour un processeur plus complexe, à savoir celui choisi pour le prototype. Cette étude ayant été réalisée sur la première version du prototype, le processeur considéré est le Leon2, doté d'un pipeline à cinq étages. Les résultats peuvent être extrapolés au cas encore plus complexe du Leon3.

Comme pour le cas du 68HC11, nous avons effectué des injections de bit-flips dans le banc de registres du processeur Leon2, pendant l'exécution de diverses applications. Outre les bit-flips simples injectés pour représenter les effets des SEUs dans des éléments de mémoire, des bits-flips multiples ont également été pris en considération. Les injections sur cet exemple plus complexe ont été réalisées grâce à l'environnement d'émulation présenté dans **[Vanh. 06]**.

Pour nos premières expérimentations, dont les résultats sont présentés dans le tableau III-III, nous avons essentiellement ciblé les registres locaux. Les registres sont classés par ordre de criticité décroissante, telle qu'évaluée par les deux approches. Notons que la comparaison des chiffres n'est pas significative, seul le classement nous intéresse.

Dans le cas du filtre FIR, nous pouvons voir que, pour ces registres locaux, les deux approches conduisent à des classements assez similaires. Globalement, les premières évaluations permettent au concepteur de prendre rapidement des décisions cohérentes au moment de décider des éléments les plus critiques à protéger. Dans le cas de Mtmx, la cohérence est plus faible, probablement en raison d'un plus grand impact des optimisations micro-architecturales pendant l'exécution du programme. Nous allons revenir sur ce point.

Tableau III-III: Taux d'erreurs obtenus après injections de fautes VS. Criticités calculées pour les applications Fir et Mtmx

	Taux d'erreurs		Criticité			Taux d'erreurs		Criticité	
FIR	%13	0.98	%13	0.75	MTMX	%16	1.00	%15	0.69
	%17	0.90	%15	0.59		%17	1.00	%16	0.60
	%15	0.79	%17	0.39		%15	0.78	%11	0.60
	%11	0.78	%14	0.37		%14	0.48	%17	0.53
	%14	0.76	%16	0.34		%11	0.38	%14	0.48
	%16	0.70	%11	0.28		%12	0.30	%10	0.26
	%12	0.70	%12	0.26		%13	0.22	%13	0.22
	%10	0.43	%10	0.21		%10	0.18	%12	0.21

Pour aller plus loin, nous avons classifié les fautes injectées dans le banc de registres, en fonction du comportement du système après l'injection, comme "Silencieuse" (pas d'effet sur les résultats de l'application), "Erreur" (mauvais résultat, mais le système continue de fonctionner et peut effectuer d'autres calculs), "Défaillance" (Mauvaise terminaison du programme, comportement inattendu, comportement futur incertain) ou "Crash" (erreur et comportement du système entièrement irrécupérable, jusqu'à la réinitialisation). Pour cette analyse, nous avons choisi une application un peu plus complexe, à savoir un chiffrement AES dont les résultats de classification sont présentés dans le tableau III-IV. Les fautes ont été injectées dans un premier temps dans l'ensemble du banc de registres, puis sélectivement dans les registres effectivement utilisés par l'application.
Un fort pourcentage d'injections de bit-flips conduit à des fautes silencieuses, même lorsque les registres ciblés sont effectivement utilisés par l'application. Les résultats de classification obtenus après injections en inversant tous les bits dans un registre sont assez similaires, bien que le nombre d'erreurs silencieuses soit un peu plus faible. Cette faible sensibilité aux erreurs dans le banc de registres a été également confirmée par d'autres campagnes d'injections effectuées sur d'autres benchmarks. Le tableau III-V résume les résultats pour Mtmx et FIR lors de l'injection de fautes sur les registres réellement utilisés par l'application. Certaines fautes silencieuses sont dues à l'écrasement des valeurs erronées, d'autres conduisent seulement à un léger retard dans le calcul. Les fautes classées comme "Défaillance" dans le filtre FIR sont principalement dues à l'avenir incertain du système.

Tableau III-IV: Classification des résultats d'injection de fautes sur le banc de registres pour l'application AES

Cibles d'injection	Tout le banc de registres	Registres utilisés dans le banc de registres
Nombre de fautes injectées (single bit-flips)	5000	1000
Marge d'erreur de la classification	1.8%	4.1%
Silencieuse	98.4%	86.4%
Erreur	0.2%	1.9%
Défaillance	0.9%	5.7%
Crash	0.5%	5.2%

Tableau III-V: Classification des résultats d'injection de fautes sur le banc de registres pour les applications MTMX et FIR

	Cibles d'injection	Registres utilisés dans le banc de registres
MTMX	Silencieuse	86,8%
	Erreur	8,8%
	Défaillance	2,8%
	Crash	1,6%
FIR	Silencieuse	87.0%
	Erreur	0%
	Défaillance	13.0%
	Crash	0%

Les effets limités de l'injection de fautes sur les registres utilisés, et ce pour les différents benchmarks, soulèvent des interrogations. En effet, la criticité des registres du banc de registres, telle qu'évaluée par les approches plus théoriques, n'est pas cohérente avec ces résultats. Nous avons donc réalisé une analyse très détaillée de l'exécution de certains programmes afin de comprendre les écarts observés entre les différentes analyses de criticité prédictives et les résultats expérimentaux par injection de fautes.

La figure 3-9 montre la structure du pipeline entier dans le microprocesseur Leon2. Le banc de registres est écrit dans le dernier des cinq étages du pipeline. Des trajets de by-pass classiques existent dans le pipeline pour la gestion des aléas d'exécution (et en particulier les dépendances de données), afin de réutiliser les valeurs dès qu'elles sont disponibles, et cela même avant leur écriture dans le banc de registres. Cette caractéristique micro-architecturale conduit à des divergences entre les évaluations faites au moment de la compilation et la criticité réelle observée lors de l'injection de fautes dans le banc de registres. En effet, dans le cas où nous avons une valeur réutilisée par la voie d'un by-pass avant d'être écrite dans le banc de registres, la corruption de cette valeur dans le banc de registres n'aura pas forcément d'impact. Quand un registre ne contient que des valeurs temporaires immédiatement réutilisées par l'instruction suivante, ce registre peut théoriquement avoir une durée de vie importante (parce que les valeurs y sont stockées très fréquemment), mais n'a dans la pratique aucune criticité, car les valeurs qu'il contient ont déjà été réutilisées par anticipation et n'auront plus de rôle dans les calculs suivants.

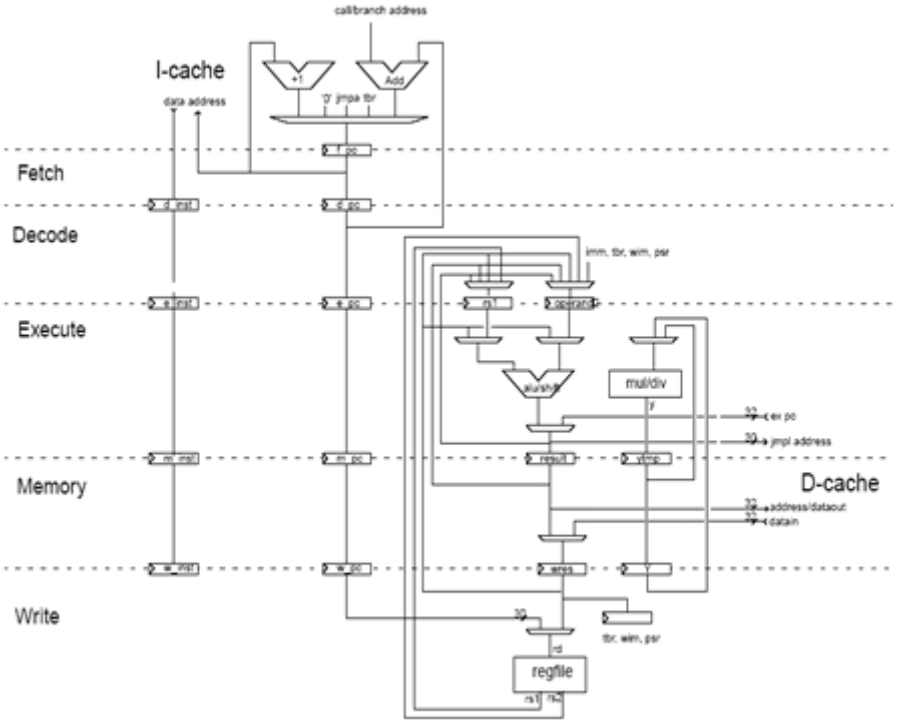


Figure 3-9 : Architecture du pipeline du microprocesseur Leon2

La conclusion de cette analyse est que l'étude de criticité des registres du banc de registres, telle qu'elle est faite dans toutes les approches existantes, ne peut être exacte que pour les architectures simples. Dans le cas contraire, la durée de vie réelle d'un registre architectural (comme ceux du banc de registres) est en fait répartie entre plusieurs registres physiques, la plupart d'entre eux étant des registres internes de la micro-architecture qui ne sont pas directement visibles par le programmeur. L'existence, dans le cas de l'architecture Sparc, d'un fenêtrage de registres, complique encore le lien entre la criticité des registres logiques et celle des registres physiques. Une étude plus spécifique pour ce type d'architecture est en cours mais

ne sera pas détaillée dans ce document. La difficulté actuelle d'assurer une bonne sélection des variables les plus critiques est l'une des raisons ayant conduit à ne pas implanter la détection des erreurs de données dans le prototype basé sur le Leon3. Toutefois, il est raisonnable de penser que le classement de criticité obtenu par notre évaluation prédictive reste utilisable au niveau des variables (ou des pseudo-registres). En effet, même si la criticité calculée se répartit entre plusieurs registres physiques, l'évaluation ne tenant pas directement compte de la micro-architecture du processeur, la somme des criticités des différents registres contenant, à un instant donné, une certaine variable peut a priori être correctement évaluée par notre approche. Ceci reste à confirmer par des études détaillées, mais ce postulat sera utilisé au chapitre 4.

III.5. Présentation de la méthode d'injection de fautes utilisée pour la validation de la méthode IDSM

III.5.1. Injection de fautes par endo-reconfiguration partielle

Pour évaluer le taux effectif de détection de la méthode IDSM, nous avons injecté des fautes sur notre prototype. Le prototype étant basé sur un FPGA Virtex V, il est possible de reconfigurer une partie de ses composants internes pendant son fonctionnement. Cette reconfiguration peut en plus être réalisée par un processeur embarqué pour réduire l'échange de données avec un PC hôte, conduisant ainsi à de meilleures performances lors de la campagne d'injection de fautes. Cette technique d'endo-reconfiguration partielle n'est de plus pas très intrusive par rapport au circuit d'origine.

L'environnement mis au point pour mener nos expériences utilise un processeur embarqué, à savoir un Microblaze, pour gérer la campagne d'injection de fautes et une IP appelée ICAP comme port de reconfiguration pour effectuer l'endo-reconfiguration partielle. Cet environnement permet l'injection d'erreurs de bit uniques ou multiples dans la configuration et nous permet de simuler le comportement d'un SEU dans les bascules utilisateur qui se trouvent dans les blocs logiques de configuration (CLB).

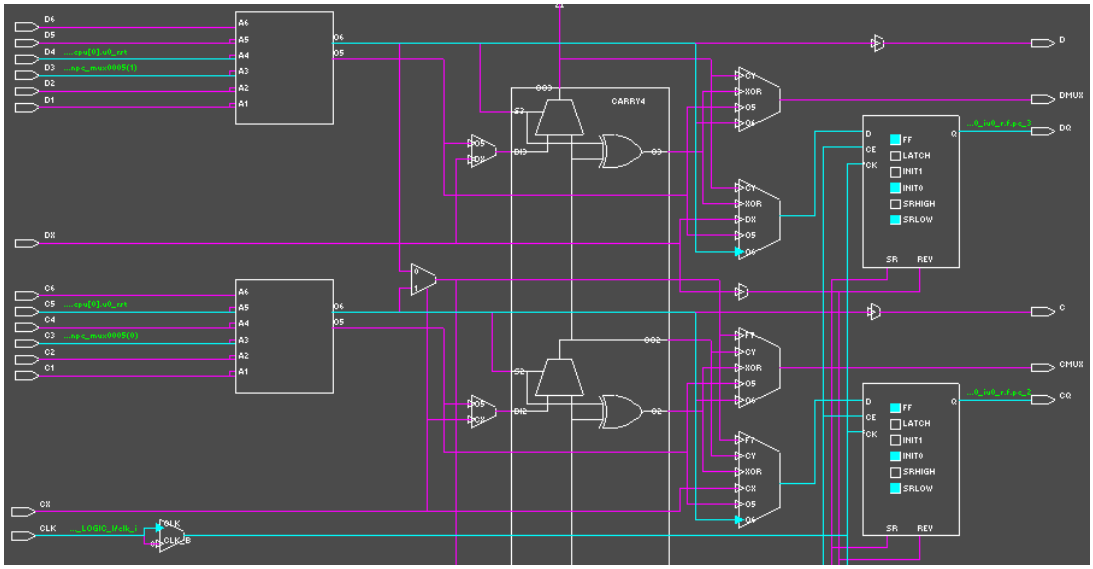
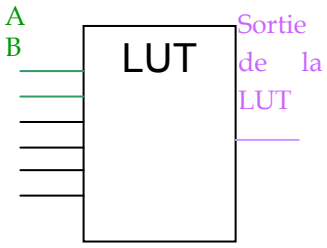


Figure 3-10 : Relation bascule/LUT

L'injection de fautes peut être déclenchée à tout moment pendant l'exécution d'une application sur le FPGA. Pour pouvoir injecter dans les registres du Leon, il faut arriver à reconfigurer les bascules le composant. Cependant, avec le Virtex-V nous ne disposons pas de toutes les informations nécessaires pour injecter directement dans le contenu des bascules. Pour émuler le comportement d'un SEU dans une bascule, nous avons donc procédé indirectement, en exploitant le fait que les bascules stockent, en général, la sortie d'une LUT, comme illustré dans la figure 3-10.

Les bits des contenus des LUTs dans les FPGA Virtex sont les bits de résultat de la table de vérité stockée en dur dans le FPGA.



(A)

Entrée		Sortie
A	B	
0	0	0
0	1	0
1	0	0
1	1	1

(B)

Index du bit de contenu	Valeur du bit de contenu	Valeur du bit de contenu après injection
0	0	1
1	0	1
2	0	1
3	1	0
4	**0**	**1**
…	**0**	**1**
63	**0**	**1**

(C)

Figure 3-11 : Description d'une LUT

La figure 3-11 illustre une LUT configurant une fonction a.b. Sur cet exemple de la figure 3-11, pour illustrer les bits de contenus de la LUT nous supposons que les deux entrées A et B sont reliées aux deux premiers signaux de la LUT. Donc la configuration va se faire dans l'ordre original de l'index de la LUT et va implémenter seulement les 4 premiers bits du contenu de la LUT (du bit 0 au bit 3), le reste des bits de contenu restant à l'état initial '0'.
Pour modifier le contenu des bascules, nous injectons donc dans le contenu des LUTs précédant les bascules visées. L'idée est la suivante : nous inversons tous les bits de contenus de façon à ce que la fonction soit la fonction inverse de celle originellement implémentée et que le résultat de sortie, qui est l'entrée de la bascule, soit un résultat erroné (Voir figure 3-11-C). Nous maintenons la configuration inverse de la fonction jusqu'à ce que le résultat de calcul se propage dans la bascule. Par exemple, en ce qui concerne le registre Fetch_PC, ceci se produit pour nos expériences dans 41% des cas au cycle suivant la reconfiguration de la LUT.

Avant une campagne d'injections, une exécution de référence de l'application exécutée par le Leon3 est faite. Celle-ci nous permet d'une part de récupérer les valeurs de bonne fin de programme qui vont nous permettre de classifier les résultats des injections. D'autre part nous générons un **`golden readback`** des contenus des bascules ciblées cycle par cycle.

Ce **`golden readback`** est l'élément de référence pour savoir quand le contenu de la bascule a été altéré par la reconfiguration de la LUT en charge du calcul, et à ce moment, nous remettons la configuration originale de la LUT. Ce flot d'injection est détaillé dans la figure 3-12.

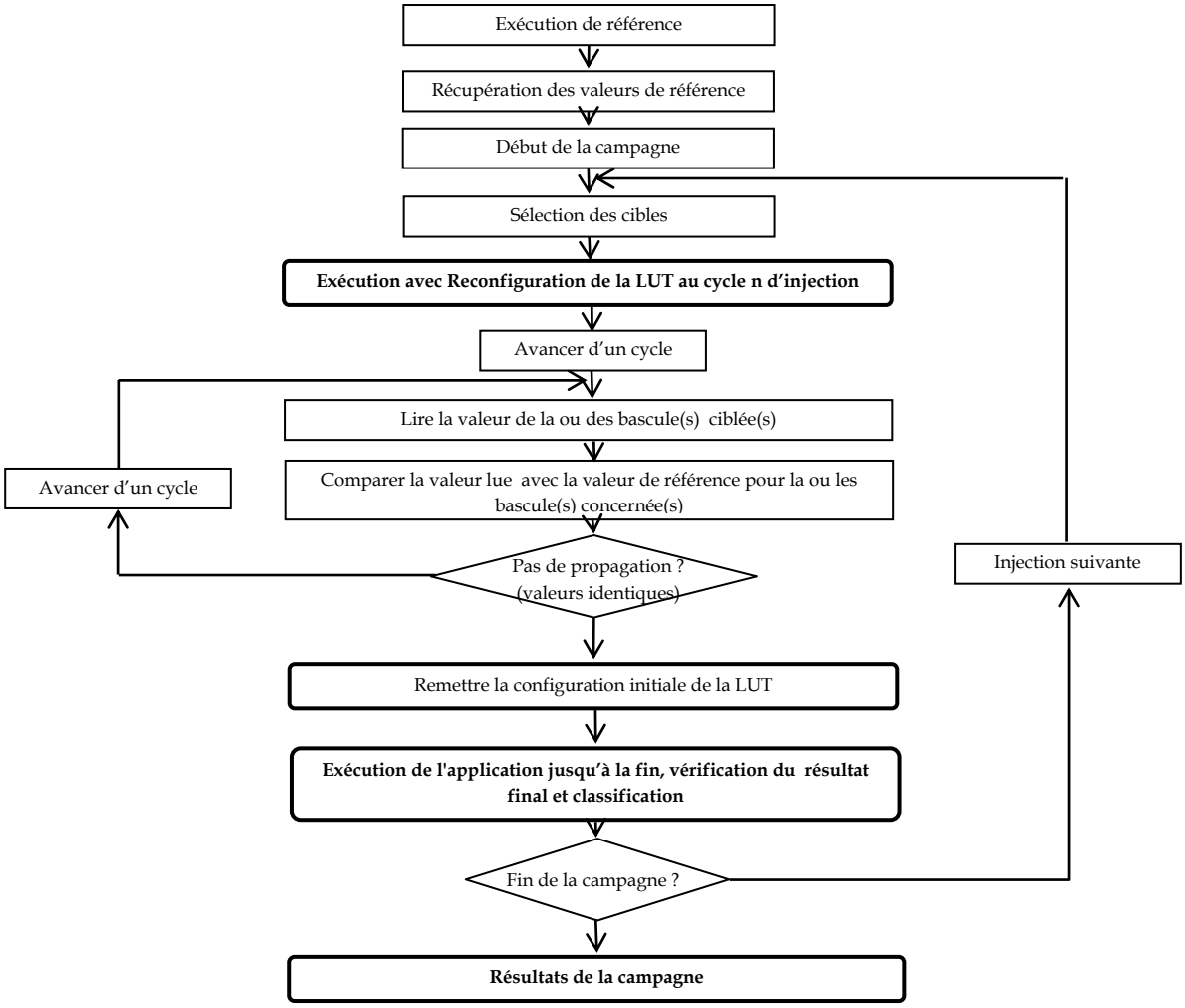


Figure 3-12 : Flot d'une campagne d'injections

III.5.2. Injection de fautes statistique: SFI:

En raison du grand nombre d'erreurs possibles qu'un circuit peut subir et la difficulté, voire l'impossibilité, de faire des injections sur toutes les cibles possibles, une sélection aléatoire d'un sous-ensemble d'erreurs est souvent incontournable en pratique.

Toutefois, le principal problème d'une telle sélection est la détermination de la confiance dans les résultats obtenus. Des études, comme celle présentée dans **[Leve. 09]**, permettent de résoudre ce problème. Plusieurs hypothèses ont été émises pour aboutir à une formule de calcul de la taille de l'échantillon. Parmi ces hypothèses : (1) Les erreurs possibles sur tous les cycles d'horloge et dans tous les éléments mémoire (Population) suivent une distribution normale ; (2) chaque erreur à chaque cycle est équiprobable, d'où la considération d'une répartition uniforme dans un échantillon ; (3) la population est finie. En tenant compte de ces hypothèses, la formule qui permet de calculer le nombre de fautes à injecter est la suivante :

$$n = \frac{N}{1 + e^2 * \frac{N-1}{t^2 * p * (1-p)}}$$

Où : n : Taille des échantillons.
N : Taille de la population.
p : Proportion d'erreur supposée
e : Marge d'erreur
t : Taux de confiance

Cette formule permet de donner le nombre n d'injections à réaliser, en fonction des autres paramètres cités dans l'équation. Le paramètre p correspond essentiellement au pourcentage d'erreurs résultant à une certaine caractéristique évaluée lors de la campagne d'injections. Comme cette valeur est a priori inconnue (mais entre 0 et 1), une approche est d'utiliser la valeur qui permettra de se placer dans le pire cas, en maximisant la taille de l'échantillon. En d'autres termes, la taille de l'échantillon sera choisie de sorte qu'elle sera forcément suffisante pour assurer (au moins) la marge d'erreur prévue avec le niveau de confiance prévu, quel que soit le résultat. Il a été démontré que ce résultat est obtenu pour p = 0,5 **[Leve. 09].** Pour la population mère, sa taille est calculée de manière à prendre en compte toutes les cibles concernées (banc de registres, mémoires…), sur l'ensemble des cycles exigés par l'application.
Par exemple, pour l'application AES, choisie pour être exécutée sur le Leon pendant les campagnes d'injection de fautes sur notre prototype, étant donné que le nombre de cycles total avant la fin de l'application est de 28237, si nous nous proposons d'injecter des MBU2, dans un registre donné de 32 bits, la taille de la population mère serait N=28237*32*(32-1).En utilisant la formule ci-dessus, nous obtenons qu'un échantillon de 400 injections nous permet de garantir une marge d'erreur de 5% et un taux de confiance de 95% (t=1,96).

III.5.3. Plan de validation de la méthode par injection de fautes

Pour valider la méthode IDSM, il faut élaborer un plan d'injection de fautes qui répond aux 3 questions suivantes : **Où** faut-il injecter ? **Quand** faut-il injecter ? et **Quel type** de fautes injecter ? Nous nous focalisons ici sur les erreurs de flot de contrôle, la vérification des données critiques n'étant pas implantée.

- **Où :** Les erreurs sur le flot de contrôle apparaissent quand le contenu d'une case mémoire ou d'un registre, stockant une instruction ou un état, est altéré. Les erreurs de flot de contrôle peuvent donc être divisées en 2 catégories :
  - Erreurs touchant l'ordre des instructions : Pour provoquer ce genre d'erreurs nous pouvons cibler les registres d'état tels que **`pc`** et **`cwp`**.
    L'injection dans le banc de registres et en particulier dans les registres **`%sp`**, **`%fp`**, **`%i7`** et **`%o7`** (contenant respectivement le pointeur de pile, le pointeur sur la fenêtre de registres, l'adresse de retour d'une fonction et l'adresse de l'instruction appelante) permet aussi de générer des erreurs de flot de contrôle.
    L'injection dans le registre d'état %icc, peut causer des erreurs de flot de contrôle mais ces erreurs ne seront pas détectables étant donné que notre méthode ne prend pas en considération les erreurs de type chemin « incorrect ».
  - Erreurs touchant le contenu des instructions : pour provoquer ce genre d'erreurs nous devons cibler soit le cache d'instructions soit les registres **`inst`** du pipeline.

Néanmoins, les caches peuvent être protégés avec des codes correcteurs donc ce ne sont pas toujours les cibles les plus pertinentes, surtout pour des erreurs de type SEU. Nous pouvons aussi viser des registres internes au processeur. Toutefois, dans ce cas et pour l'approche générique, l'instruction lue par le processeur et le watchdog est la bonne ; une erreur sur le code instruction survenant après cette lecture ne pourra être détectée par le watchdog qu'indirectement, suite à une erreur de flot de contrôle (une simple erreur de calcul ne pourra pas être détectée). Dans le cas de notre prototype, il a été signalé que les signaux lus par le watchdog sont situés en entrée de l'étage de décodage. Injecter dans le registre **`inst`** en entrée de l'étage de décodage revient donc en fait, par rapport à l'approche générique, à injecter dans le cache instructions, c'est-à-dire que l'erreur peut être vue directement par le watchdog. Il serait possible d'injecter dans des registres situés plus profondément dans le pipeline pour étudier les cas où le watchdog lit une instruction non perturbée. Nous nous limiterons dans la suite au premier cas. D'une part, aucune protection n'est implantée dans la structure de caches. D'autre part, nous nous intéresserons surtout à des erreurs multiples, qui ne pourraient pas être correctement gérées par les protections habituelles (parité ou ECC). L'étude du premier cas est donc pertinente ; celle du second cas fait partie des études futures.

Le choix a été fait de se limiter à des injections dans le système initial, et notamment dans le processeur Leon3 car les erreurs les plus critiques sont celles pouvant perturber l'application. Des erreurs peuvent aussi survenir dans le watchdog ou ses mémoires mais, soit elles seraient détectées, soit leur conséquence serait au pire limitée à une suppression de la surveillance, sans impact direct sur le déroulement immédiat de l'application. Ce cas n'a donc pas été analysé jusque là, mais pourrait aussi faire l'objet de campagnes ultérieures.

- **Quand :** Comme nous l'avons vu dans le paragraphe II-6, le taux de couverture théorique des erreurs n'est pas le même selon le moment d'injection. En effet, les instructions ajoutées après l'édition des liens et donc relatives au boot et aux fonctions de bibliothèques ne sont pas contrôlées de la même manière que le reste du programme, faute de disponibilité des sources. Il faut, par conséquent, réaliser deux campagnes distinctes : une première campagne pendant le boot logiciel et une deuxième campagne pendant l'exécution du programme principal. Même si le contrôle des fonctions de bibliothèque s'apparente plus au contrôle pendant le boot logiciel, nous allons considérer ces fonctions dans le programme principal par soucis de simplification des campagnes d'injection. Cette classification n'aura pas d'incidences significatives sur les résultats d'injections dans le registre **`PC`**, mais peut affecter le taux de détection dans le registre **`inst`**.
  Il est possible également de faire des injections pendant le boot matériel du Leon. En effet, même si pendant cette phase le watchdog n'est pas encore démarré, il serait intéressant de voir s'il y aura détection des fautes dont les effets se prolongent au-delà du boot matériel.

- **Quel type** : Les méthodes de protection classiques permettent en général de détecter les erreurs de multiplicité simple. La méthode IDSM doit permettre en priorité la détection de fautes de multiplicité plus élevée. Dans les campagnes que nous allons réaliser, nous injecterons donc des fautes de multiplicité allant de 1 à 5. Nous avons fait le choix de ne pas choisir des multiplicités plus élevées, car outre l'augmentation des temps expérimentaux, la probabilité de multiplicités plus grandes, surtout au niveau d'un même registre, semble assez faible dans la plupart des situations concrètes.

III.5.4. Classification des fautes injectées

A la fin de l'application AES, une comparaison du résultat obtenu avec une valeur de référence est effectuée pour déterminer si le chiffrement s'est bien déroulé. Si le résultat est conforme aux attentes, le programme fait un appel à la fonction success() ; dans le cas contraire, c'est la fonction failure() qui est appelée. Il existe donc 3 issues possibles, après injection de fautes : success(), failure() ou aucune de ces deux fonctions n'est appelée. Les deux derniers cas correspondent respectivement aux classes « erreur de données » et « crash ».
Cependant, le programme peut aboutir dans certains cas à des résultats corrects et donc à l'appel à la fonction success() mais avec un certain délai (positif ou négatif) par rapport à l'instant nominal de terminaison. Même si le watchdog, dans sa conception actuelle, n'a aucun moyen de détecter les erreurs de délai avec une sortie correcte, les fautes injectées engendrant ce genre de situation ne seront pas considérées comme des fautes silencieuses. Elles seront donc regroupées avec les crashs dans la catégorie « Non terminaison au bon cycle ». De cette façon, nous allons nous placer dans le pire cas, à savoir un contexte d'application temps réel strict, sans même accepter le cas des terminaisons anticipées.

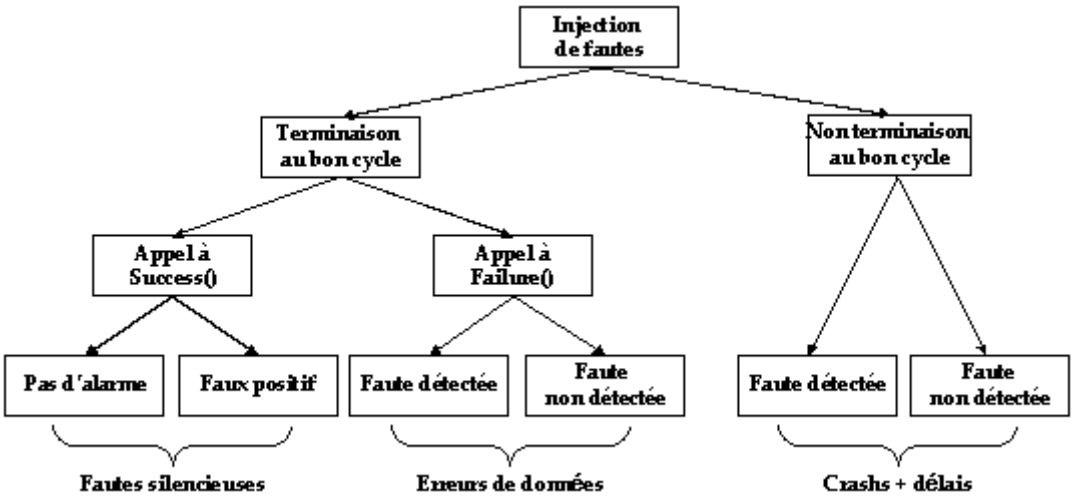


Figure 3-13 : Classification des fautes injectées

Outre les cas typiques de terminaison, le Leon3 peut présenter quelques dysfonctionnements après injection de fautes. En effet, dans quelques cas, le Leon3 saute à une mauvaise adresse, puis à une adresse dans la trap table et pour finir stoppe.
Le problème est qu'il stoppe avant que le watchdog n'ait pu s'apercevoir de ce dysfonctionnement à cause de l' « annul_bit » du Leon3 qui est mis à '1'. Or ce bit, comme nous

l'avons expliqué dans le paragraphe III.3.1 apparaît dans l'étage des exceptions du Leon. S'il est levé alors l'instruction étant dans le même étage n'est pas exécutée. C'est pourquoi le watchdog ne prend pas en compte ces instructions, se fige, attend que le Leon3 "se réveille" et le signal d'alarme n'est pas levé. Ces comportements "anormaux" du processeur seront identifiés et comptabilisés pour avoir une idée sur leur pourcentage. Par ailleurs, dans notre classification, ils seront considérés, comme les crashs et les délais, comme des cas de non terminaison au bon cycle, même si nous savons d'avance qu'ils ne seront pas signalés par une alarme, et donc classés comme non détectés. Ces cas sont toutefois assez faciles à détecter en ajoutant une fonction timer au watchdog.

III.6. Analyse de l'efficacité de la méthode IDSM

III.6.1. Erreurs touchant l'ordre des instructions

III.6.1.1. Résultats d'injections sur le Fetch-PC

Nous avons injecté des fautes de multiplicité allant de 1 à 5 sur le registre Fetch-PC du pipeline du Leon. Les résultats obtenus pendant cette campagne sont détaillés dans le tableau III-VI.

Tableau III-VI : Résultats d'injections sur le Fetch-PC

		Terminaison au bon cycle				**Non Terminaison au bon cycle**		
		Appel à Success()		**Appel à Failure ()**				
		Pas d'alarme	**Faux positifs**	**Détecté**	**Non détecté**	**Détecté**	**Non détecté**	**% de détection**
MBU5	Boot matériel	5	0	0	0	5	35	**12,50%**
	Boot Logiciel	6	1	0	0	72	31	**69,90%**
	Pgm Principal	2	0	0	0	342	74	**82,21%**
	Total	13	1	0	0	419	140	**74,96%**
MBU4	Boot matériel	3	0	0	0	5	22	**18,52%**
	Boot Logiciel	2	1	0	0	69	30	**69,70%**
	Pgm Principal	2	0	0	0	375	64	**85,42%**
	Total	7	1	0	0	449	116	**79,47%**
MBU3	Boot matériel	7	0	0	0	1	31	**3,13%**
	Boot Logiciel	4	1	0	0	73	29	**71,57%**
	Pgm Principal	4	0	0	0	374	49	**88,42%**
	Total	15	1	0	0	448	109	**80,43%**
MBU2	Boot matériel	11	0	0	0	2	34	**5,56%**
	Boot Logiciel	14	0	0	0	83	25	**76,85%**
	Pgm Principal	15	0	0	0	358	31	**92,03%**
	Total	40	0	0	0	443	90	**83,11%**
SEU	Boot matériel	8	0	0	0	0	26	**0,00%**
	Boot Logiciel	12	0	0	0	86	16	**84,31%**
	Pgm Principal	31	0	0	0	362	32	**91,88%**
	Total	51	0	0	0	448	74	**85,82%**

Le meilleur taux de détection, pour toutes les multiplicités, est obtenu bien sûr pendant le programme principal, jusqu'à 92,03% pour des fautes MBU2. Quelques fautes injectées pendant le boot matériel ont même été détectées, bien que le watchdog ne soit pas opérationnel pendant cette phase. Ceci vient essentiellement du fait que les effets de ces fautes se sont prolongés au delà de cette phase, atteignant la phase de boot logiciel. Nous enregistrons pour la multiplicité 4 le meilleur taux de détection dans le boot matériel à savoir 18,52%. Par ailleurs, nous remarquons que plus la multiplicité augmente plus le taux de détection global diminue, entre 74,96% pour les MBU5 et 85,82% pour les SEU. Toutefois, le taux de détection reste élevé même pour de grandes multiplicités.

Aucun appel à la fonction failure() n'a été enregistré au cycle 28237. Ceci s'explique par le fait qu'une altération du comportement du Leon pendant le calcul s'accompagne généralement d'un délai et est donc comptabilisée parmi les cas de non terminaison au bon cycle.
Pour comprendre mieux le taux de non terminaison au bon cycle non détecté, nous avons testé en simulation certaines injections ayant conduit à ce cas. Nous avons trouvé que la grande majorité, si ce n'est la totalité, de ces injections provoquaient soit des dysfonctionnements du Leon3 soit des erreurs de délais (à un degré moindre). Nous avons donc rajouté une classification qui dépend de l'annul_bit et de l'adresse du Leon. Ces cas de dysfonctionnements et de délais parmi les fautes non détectées sont reportés dans le tableau III-VII.

Tableau III-VII : Répartition des cas de non détection parmi les fautes injectées sur le Fetch-PC ayant conduit à une non terminaison au bon cycle

		Délai		Dysfonctionnement du Leon3	Crash
		Failure()	Success()		
MBU5	Boot matériel	0%	28,57%	2,86%	68,57%
	Boot Logiciel	0%	3,23%	54,84%	41,94%
	Pgm Principal	0%	1,35%	52,70%	45,95%
	Total	0%	8,57%	40,71%	50,71%
MBU4	Boot matériel	0%	27,27%	0%	72,73%
	Boot Logiciel	0%	6,67%	50,00%	43,33%
	Pgm Principal	3,13%	0%	51,56%	45,31%
	Total	1,72%	6,90%	41,38%	50,00%
MBU3	Boot matériel	0%	35,48%	0,00%	64,52%
	Boot Logiciel	0%	10,34%	55,17%	34,48%
	Pgm Principal	0%	4,08%	51,02%	44,90%
	Total	0%	14,68%	37,61%	47,71%
MBU2	Boot matériel	2,94%	20,59%	0,00%	76,47%
	Boot Logiciel	0,00%	16,00%	40,00%	44,00%
	Pgm Principal	19,35%	16,13%	16,13%	48,39%
	Total	7,78%	17,78%	16,67%	57,78%
SEU	Boot matériel	7,69%	38,46%	3,85%	50,00%
	Boot Logiciel	0,00%	62,50%	25,00%	12,50%
	Pgm Principal	12,50%	34,38%	34,38%	18,75%
	Total	8,11%	41,89%	21,62%	28,38%

Les dysfonctionnements dépassent 50% des injections non détectées pendant le boot logiciel et l'exécution du programme principal pour les multiplicités allant de 3 à 5, mais ils n'expliquent pas à eux seuls le taux de non détection. Pour cela nous avons effectué des simulations de quelques fautes ayant engendré des cas de non détection. Nous avons remarqué une récurrence des bits 30 et 31 dans les cibles des fautes dont la modification provoque l'arrêt net du Leon. Pour l'instant, ces cas sont donc répertoriés comme des crashs alors que l'ajout d'un timer permettrait de les détecter.
Si nous nous intéressons aux erreurs de données (Appel à la fonction failure() retardé ou non) et aux crashs, nous obtenons le tableau III-VIII. Comme pour la détection globale, plus la multiplicité diminue, plus le taux de détection des crashs augmente atteignant 98,24% dans le programme principal en SEU. Toutefois, ce taux pourrait atteindre 100% pour toutes les multiplicités avec l'ajout d'une fonction timer à notre watchdog.

Tableau III-VIII: Taux de détection des erreurs de données et des crashs sur le Fetch-PC

		Erreur de données (failure())			Crash		
		Détecté	**Non détecté**	**% Détection**	**Détecté**	**Non détecté**	**% Détection**
MBU5	Boot matériel	0	0	-	5	24	17,24%
	Boot Logiciel	0	0	-	72	13	84,71%
	Pgm Principal	0	0	-	342	34	90,96%
	Total	0	0	-	419	71	85,51%
MBU4	Boot matériel	0	0	-	5	16	23,81%
	Boot Logiciel	0	0	-	69	13	84,15%
	Pgm Principal	1	2	33,33%	373	29	92,79%
	Total	1	2	33,33%	447	58	88,51%
MBU3	Boot matériel	1	0	100%	0	20	0%
	Boot Logiciel	0	0	-	72	10	87,80%
	Pgm Principal	2	0	100%	372	22	94,42%
	Total	3	0	100%	444	52	89,52%
MBU2	Boot matériel	0	1	0%	2	26	7,14%
	Boot Logiciel	3	0	100%	79	11	87,78%
	Pgm Principal	4	6	40%	352	15	95,91%
	Total	7	7	50%	433	52	89,28%
SEU	Boot matériel	0	2	0%	0	13	0%
	Boot Logiciel	0	0	-	82	2	97,62%
	Pgm Principal	11	4	73,33%	335	6	98,24%
	Total	11	6	64,71%	417	21	95,21%

Le nombre de faux positifs, répertoriés dans le tableau III-VI, est assez important pour les multiplicités de 3 à 5. Dans 42,85% des cas ils se présentent pendant l'exécution du programme principal. Une étude des traces des injections a montré que la modification des bits 26->29 du registre PC n'a aucun effet sur le comportement du Leon. Le tableau III-IX présente le taux des cas de faux positifs par rapport au nombre total des fautes silencieuses. La figure 3-14 illustre un cas de faux positif, généré à cause de l'insensibilité du Leon aux fautes dans le bit 26 du PC. En effet, en comparant les adresses des instructions successives, le watchdog se rend compte

que les adresses ne sont pas adjacentes et émet une alarme. Alors que le Leon exécute l'instruction correspondant à l'adresse 40001724 au lieu de celle à l'adresse fautée 440001724.

Tableau III-IX : Proportion des faux positifs dans les campagnes sur le Fetch-PC

	Fautes silencieuses	**Faux positifs**	**% faux positifs par rapport au total des fautes silencieuses**
MBU5	13	1	7,14%
MBU4	7	1	12,50%
MBU3	15	1	6,25%
MBU2	40	0	0%
SEU	51	0	0%

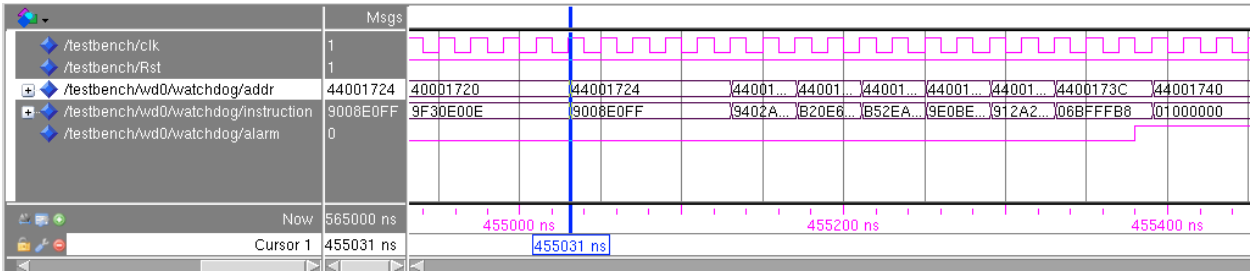


Figure 3-14 : Exemple de fautes dans le Fetch-PC engendrant un faux positif

III.6.1.2. Résultats d'injections sur le CWP

Dans l'architecture Sparc le nombre de fenêtres de registres varie entre 2 et 32 selon l'implémentation. Dans notre version du Leon3, il existe seulement 8 fenêtres, comme illustré dans la figure 3-15. Le registre CWP est donc composé de 3 bits seulement, c'est la raison pour laquelle nous allons y injecter des SEUs.

A partir des résultats présentés dans le tableau III-X, nous remarquons qu'aucune injection pendant le boot matériel n'a eu d'effet. Ceci s'explique par le fait que pendant cette phase, le mécanisme de fenêtrage n'est pas encore utilisé. D'ailleurs, pendant une exécution nominale de AES sans injection de fautes, le CWP garde la même valeur du reset jusqu'au début du boot logiciel. Par contre, les injections pendant les phases de boot logiciel et du programme principal conduisent dans 99,4% à une altération du comportement du Leon.

Tableau III-X : Résultats d'injections de fautes dans le CWP

	Terminaison au bon cycle				**Non Terminaison au bon cycle**		
	Appel à Success()		**Appel à Failure ()**				
	Pas d'alarme	**Faux positifs**	**Détectée**	**Non détectée**	**Détectée**	**Non détectée**	**% de détection**
Boot matériel	30	0	0	0	0	0	-
Boot Logiciel	0	0	0	0	113	11	**91,13%**
Pgm Principal	4	0	0	0	486	62	**88,69%**
Total	34	0	0	0	599	73	**89,14%**

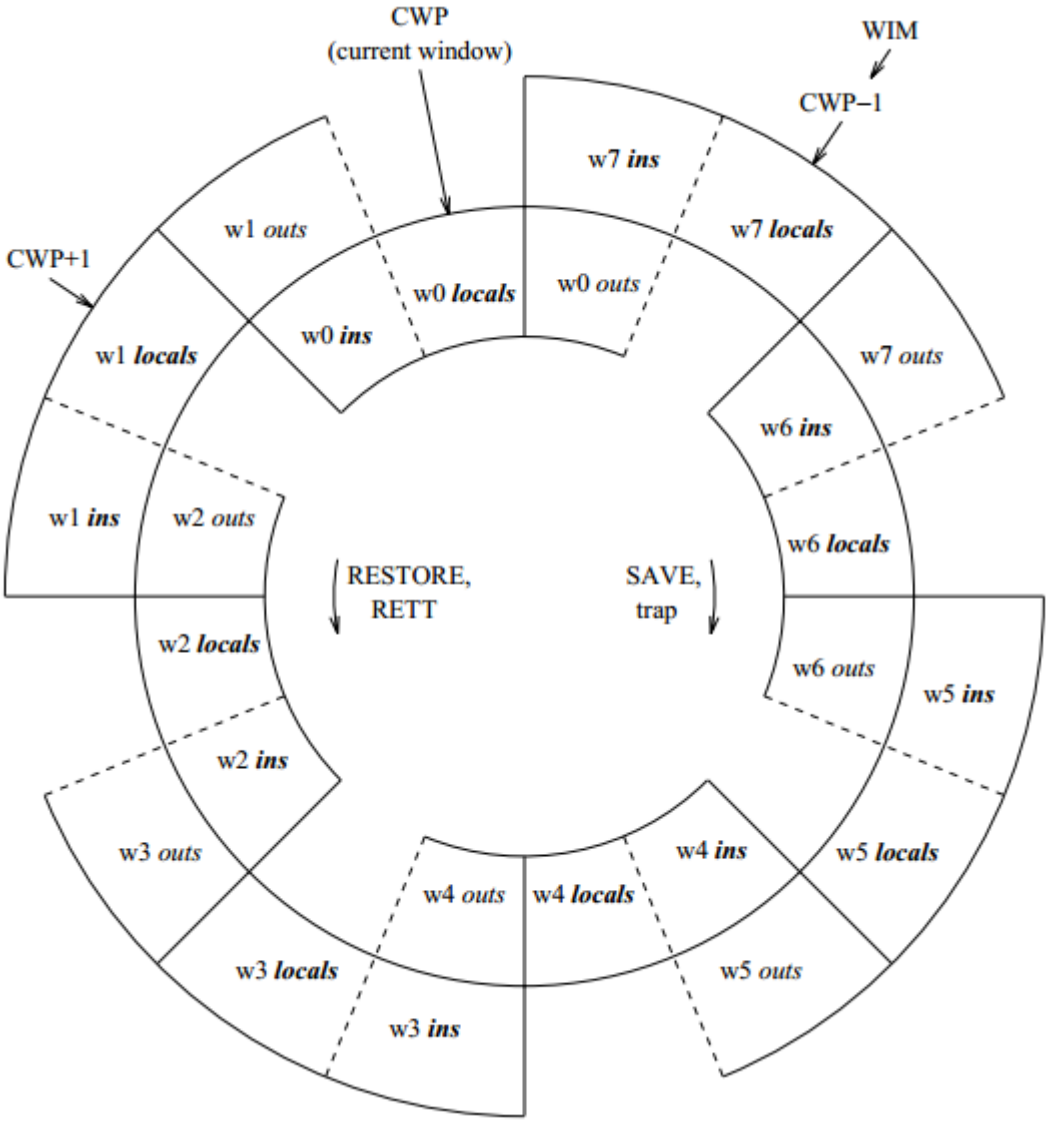


Figure 3-15 : Fenêtrage des registres dans le Leon3

III.6.1.3. Résultats d'injections sur le banc de registres

Il existe 136 registres physiques dans notre version du Leon 3 dont 8 registres généraux et 128 registres distribués sur 8 fenêtres possibles. La figure 3-16 donne la correspondance entre les noms logiques des registres et leurs numéros dans le banc de registres.

Dans notre prototype, le banc de registres est implanté dans des LUTs configurées en RAM. Par conséquent, la méthode d'injection de fautes présentée dans le paragraphe III.5.1, et permettant de modifier indirectement le contenu des bascules d'un registre par altération des LUTs les précédant, n'a pas lieu d'être utilisée puisque nous pouvons injecter directement dans les LUTs. Pour les injections de fautes dans le banc de registres nous allons donc procéder comme suit, étant donné que chaque LUT est constituée de 64 bits de contenu, et engloberait donc deux

registres. Pour injecter une faute, nous choisissons aléatoirement le cycle et la cible ; celle-ci est une constitué d'un triplet {coordonnées d'une LUT, un registre, index du bit dans le registre}. Etant donné que le Leon 3 utilise le système de fenêtrage, il est compliqué d'identifier à chaque cycle, pendant les injections, les registres ayant un effet sur le flot de contrôle à savoir les registres **`%sp`**, **`%fp`**, **`%i7`** et **`%o7`** (contenant respectivement le pointeur sur la pile, le pointeur sur la frame de registres, l'adresse de retour d'une fonction et l'adresse de l'instruction appelante) parmi tous les autres registres. Nous avons donc injecté des MCU5 sur tout le banc de registres sauf les registres généraux. La probabilité de tomber sur les registres **`%sp`**, **`%fp`**, **`%i7`** et **`%o7`** est alors égale à 4/(32-8) =16,66%.

		registre physique :							
# Register	Register name	cwp = 0	cwp = 1	cwp = 2	cwp = 3	cwp = 4	cwp = 5	cwp = 6	cwp = 7
0	%g0	128	128	128	128	128	128	128	128
1	%g1	129	129	129	129	129	129	129	129
2	%g2	130	130	130	130	130	130	130	130
3	%g3	131	131	131	131	131	131	131	131
4	%g4	132	132	132	132	132	132	132	132
5	%g5	133	133	133	133	133	133	133	133
6	%g6	134	134	134	134	134	134	134	134
7	%g7	135	135	135	135	135	135	135	135
8	%o0	8	24	40	56	72	88	104	120
9	%o1	9	25	41	57	73	89	105	121
10	%o2	10	26	42	58	74	90	106	122
11	%o3	11	27	43	59	75	91	107	123
12	%o4	12	28	44	60	76	92	108	124
13	%o5	13	29	45	61	77	93	109	125
14	%o6 - %sp	14	30	46	62	78	94	110	126
15	%o7	15	31	47	63	79	95	111	127
16	%l0	16	32	48	64	80	96	112	0
17	%l1	17	33	49	65	81	97	113	1
18	%l2	18	34	50	66	82	98	114	2
19	%l3	19	35	51	67	83	99	115	3
20	%l4	20	36	52	68	84	100	116	4
21	%l5	21	37	53	69	85	101	117	5
22	%l6	22	38	54	70	86	102	118	6
23	%l7	23	39	55	71	87	103	119	7
24	%i0	24	40	56	72	88	104	120	8
25	%i1	25	41	57	73	89	105	121	9
26	%i2	26	42	58	74	90	106	122	10
27	%i3	27	43	59	75	91	107	123	11
28	%i4	28	44	60	76	92	108	124	12
29	%i5	29	45	61	77	93	109	125	13
30	%i6 - %fp	30	46	62	78	94	110	126	14
31	%i7	31	47	63	79	95	111	127	15

Figure 3-16 : Correspondance entre les noms logiques des registres et leurs numéros dans le banc de registres en fonction du CWP

Tableau III-XI : Résultats d'injections de fautes dans le banc de registres

	Terminaison au bon cycle				Non Terminaison au bon cycle		
	Appel à Success()		Appel à Failure ()				
	Pas d'alarme	Faux positifs	Détectée	Non détectée	Détectée	Non détectée	% de détection
Boot matériel	253	0	0	0	0	0	-
Boot Logiciel	713	0	0	0	2	18	**10,00%**
Pgm Principal	2344	0	0	0	90	509	**15,03%**
Total	3310	0	0	0	92	527	**14,86%**

Le taux de détection des fautes injectées est de 14,86% et atteint 15,03% pendant l'exécution du programme principal. Etant donné que seules 16,66% des injections peuvent atteindre les registres affectant le flot de contrôle, une estimation du taux de détection des erreurs dans les registres **`%sp`**, **`%fp`**, **`%i7`** et **`%o7`** atteint les 90%.

Tout comme pour les injections dans le CWP, les injections pendant le boot matériel n'ont eu aucun effet. Par ailleurs, aucun cas de faux positif ni d'appel à la fonction failure() n'a été signalé.

Nous avons cherché à savoir combien de cas de délais conduisent à une non terminaison au bon cycle. Nous avons considéré à titre indicatif un délai de +/- 1600 cycles sachant que dans les conditions normales AES nécessite 28237 cycles. D'après le tableau III-XII, plus d'un tiers des cas de non terminaison au bon cycle correspondent à un délai avec un appel à la fonction Success() qui garantit le bon déroulement du calcul.

Tableau III-XII : Proportion des délais par rapport au total des fautes non détectées

	Non détecté	Retard +/-1600 cycles avec Success()	% retard par rapport aux non détectés
Boot matériel	0	0	-
Boot Logiciel	18	7	38,89%
Pgm Principal	509	184	36,15%
Total	527	191	36,24%

III.6.2. Erreurs touchant le contenu des instructions

Nous avons injecté des fautes de multiplicité 1, 3, 4 et 5 sur le registre Decode-inst du pipeline du Leon. La classification des fautes injectées pendant cette campagne est détaillée dans le tableau III-XIII.

Tableau III-XIII : Résultats des injections dans le Decode-Inst

		Terminaison au bon cycle				Non Terminaison au bon cycle		
		Appel à Success()		Appel à Failure ()				
		Pas d'alarme	Faux positifs	Détectée	Non détectée	Détectée	Non détectée	% de détection
SEU	Boot matériel	23	0	0	0	0	6	**0,00%**
	Boot Logiciel	46	1	0	0	20	28	**41,67%**
	Pgm Principal	134	74	0	0	114	60	**65,52%**
	Total	203	75	0	0	134	94	**58,77%**
MBU3	Boot matériel	25	0	0	0	0	12	**0,00%**
	Boot Logiciel	25	0	0	0	15	32	**31,91%**
	Pgm Principal	66	52	0	0	181	88	**67,29%**
	Total	116	52	0	0	196	132	**59,76%**
MBU4	Boot matériel	17	0	0	0	1	10	**9,09%**
	Boot Logiciel	31	1	0	0	20	58	**25,64%**
	Pgm Principal	23	32	1	0	231	73	**75,99%**
	Total	71	33	1	0	252	141	**64,12%**
MBU5	Boot matériel	18	0	0	0	1	22	**4,35%**
	Boot Logiciel	16	0	0	0	54	81	**40,00%**
	Pgm Principal	53	52	1	0	450	99	**81,97%**
	Total	87	52	1	0	505	202	**71,43%**

Tout comme les injections dans le Fetch-PC, quelques fautes injectées pendant le boot matériel ont été détectées, et le taux de détection atteint 9,04% dans le cas des MBU4. Le meilleur taux de détection, pour toutes les multiplicités, est obtenu pendant le programme principal, jusqu'à 81,97% pour MBU5. Par ailleurs, le taux de détection pendant le boot logiciel dépasse nettement le taux attendu. En effet, le contrôle du contenu des instructions relatives à cette phase se résume à la vérification des 4 premiers bits de chaque instruction, ce qui correspond à un taux de détection théorique égal à 12,5%. Ceci peut être expliqué par le fait qu'à part les 4 bits vérifiés, le watchdog peut détecter la faute si les bits touchés ont une incidence sur le flot de contrôle. Cette différence avec le taux de détection théorique est donc liée davantage à l'effet fonctionnel de la modification du code d'instruction, qu'au nombre de bits vérifiés.

Le taux d'injections ayant conduit à des cas de faux positifs atteint 14,82% pour les SEU, ce qui peut être expliqué par le fait que la faute touche un bit non utilisé de l'instruction. Mais plus la multiplicité augmente, plus ce taux diminue.
La figure 3-17 illustre un cas d'injection de faute dans le troisième bit du registre Decode-inst ayant engendré un faux positif. Le watchdog compacte l'instruction fautée et déclenche une alarme, alors que le Leon3 continue normalement son exécution.

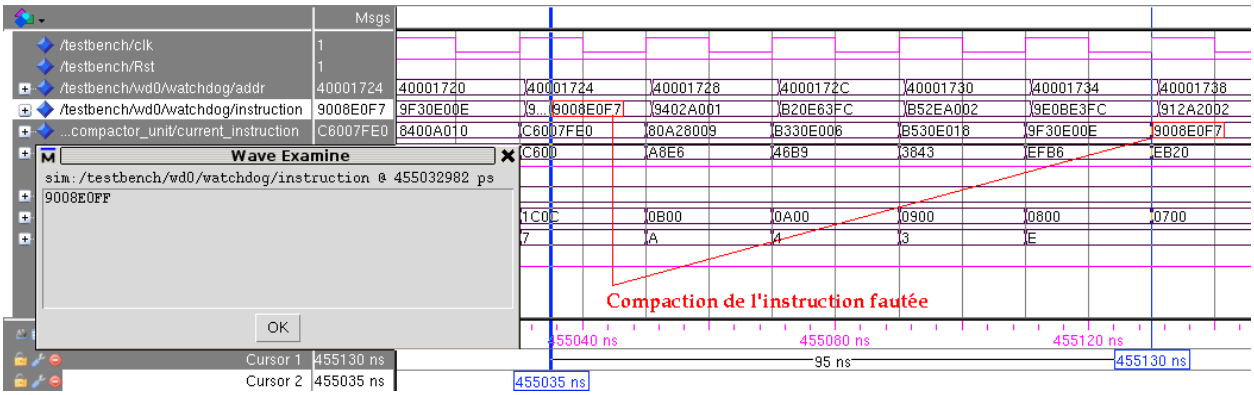


Figure 3-17 : Exemple de fautes dans le Decode-Inst engendrant un faux positif

Pour mieux comprendre les raisons de non détection parmi les cas de non terminaison au bon cycle, nous avons déterminé les proportions de crashs, de délais et de dysfonctionnements parmi ces cas. Tout comme pour les campagnes précédentes, nous avons considéré à titre indicatif un délai de +/-1600 cycles. Les délais supérieurs à cet intervalle ont été considérés comme des crashs.

Tableau III-XIV : Répartition des cas de non détection parmi les fautes injectées sur le Decode-Inst ayant conduit à une non terminaison au bon cycle

		Délai		Dysfonctionnement du Leon3	Crash
		Failure()	Success()		
MBU5	Boot matériel	0%	40,91%	0%	59,09%
	Boot Logiciel	0%	30,86%	51,85%	17,28%
	Pgm Principal	0%	6,06%	86,87%	7,07%
	Total	0%	19,80%	63,37%	16,83%
MBU4	Boot matériel	0%	60,00%	10,00%	30,00%
	Boot Logiciel	0%	18,97%	67,24%	13,79%
	Pgm Principal	0%	5,48%	90,41%	4,11%
	Total	0%	14,89%	75,18%	9,93%
MBU3	Boot matériel	0%	58,33%	8,3%	33,33%
	Boot Logiciel	0%	34,38%	62,50%	3,13%
	Pgm Principal	0%	3,41%	90,91%	5,68%
	Total	0%	15,9%	76,52%	7,58%
SEU	Boot matériel	0%	50,00%	0%	50,00%
	Boot Logiciel	0%	50,00%	35,71%	14,29%
	Pgm Principal	5%	15,00%	75,00%	5,00%
	Total	3,19%	27,66%	58,51%	10,64%

D'après le tableau III-XIV, les dysfonctionnements du Leon3 atteignent 90% des cas de non détection pendant le programme principal, en MBU3 et MBU4. Ces dysfonctionnements sont survenus essentiellement pendant le boot logiciel et l'exécution du programme principal, leur taux atteint 35% du total des fautes injectées en MBU4 pendant le boot logiciel. Par ailleurs, entre 15% et 27% des cas non détectés de non terminaison au bon cycle correspondent à un délai avec un appel à la fonction Success() qui garantit le bon déroulement du calcul.

Si nous nous focalisons donc sur les erreurs de données (Appel à la fonction failure() retardé ou non) et sur les crashs, nous obtenons le tableau III-XV.

Tableau III-XV : Taux de détection des erreurs de données et des crashs sur le Decode-Inst

		Erreur de données (failure())			Crash		
		Détecté	Non détecté	% Détection	Détecté	Non détecté	% Détection
MBU5	Boot matériel	0	0	-	1	13	**7,14%**
	Boot Logiciel	0	0	-	51	14	**78,46%**
	Pgm Principal	51	0	**100%**	343	7	**98,00%**
	Total	51	0	**100%**	395	34	**92,07%**
MBU4	Boot matériel	0	0	-	1	3	**25,00%**
	Boot Logiciel	0	0	-	17	8	**68,00%**
	Pgm Principal	21	0	**100%**	173	3	**98,30%**
	Total	21	0	**100%**	191	14	**93,17%**
MBU3	Boot matériel	0	0	-	0	4	**0,00%**
	Boot Logiciel	0	0	-	14	1	**93,33%**
	Pgm Principal	18	0	**100%**	126	5	**96,18%**
	Total	18	0	**100%**	140	10	**93,33%**
SEU	Boot matériel	0	0	-	0	3	**0,00%**
	Boot Logiciel	0	0	-	17	4	**80,95%**
	Pgm Principal	27	3	**90,00%**	66	3	**95,65%**
	Total	27	3	**90,00%**	83	10	**89,25%**

III.7. Evaluation des coûts de la méthode IDSM

Comme toutes les méthodes de vérification de flot de contrôle à signature disjointe, la méthode IDSM, ne nécessitant pas la modification du code applicatif, présente un coût en performance nul. Dans ce paragraphe, on se propose de déterminer les autres surcoûts que peut présenter notre méthode à savoir les surcoûts en surface et en mémoire.

III.7.1. Evaluation des surcoûts en surface de la méthode IDSM

Deux versions du watchdog ont été synthétisées en optimisant la vitesse et en gardant la hiérarchie, la première en gardant la ROM dans la description du watchdog et la deuxième en l'externalisant. Cette ROM contient le programme du watchdog pour l'application AES, telle qu'utilisée sur le prototype.

Par ailleurs, la version réduite leon3mp Leon a été synthétisée, en optimisant la surface et sans garder la hiérarchie. Voici les principales caractéristiques de cette version :

- Pas de fonctionnalités de tolérance aux fautes,
- Nombre de CPU = 1,
- Pas de FPU (Floating Point Unit),
- Eléments non implémentés : DSU Ethernet, contrôleur PROM/SRAM, contrôleur SDRAM, AHB ROM, AHB RAM, core Ethernet de Gaisler, interface CAN 2.0, interface PCI, interface Spacewire.

Les taux d'occupation du FPGA Virtex V sont illustrés dans la figure 3-18. Les résultats détaillés du nombre de bascules, LUTs et blocs RAM utilisés sont présentés dans les tableaux III-XVI et

III-XVII pour le watchdog sans et avec ROM respectivement. Le nombre de bascules utilisées pour le watchdog reste approximativement le même entre les deux versions du watchdog (avec et sans la ROM). Ce nombre représente le quart de ce qui est consommé pour l'implémentation du Leon3mp. Par ailleurs, le nombre de LUTs utilisées comme logique a plus que doublé entre les deux versions du watchdog sans pour autant dépasser 70% des LUTs utilisées pour le Leon3. Bien sûr, la taille de la ROM dépend de l'application à surveiller.

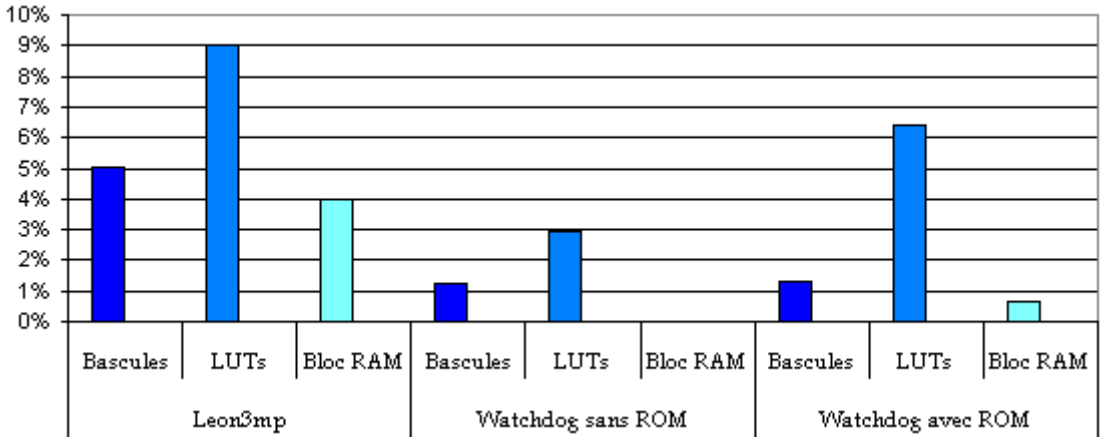


Figure 3-18 : Taux d'occupation du FPGA Virtex V par le Leon3 et les deux versions du watchdog

Tableau III-XVI : Caractéristiques du watchdog (sans ROM) par rapport au Leon3

	Leon3mp		Watchdog		Watchdog/Leon3mp ratio
	nb	% Virtex5	nb	% Virtex5	
Bascules	3503	5%	862	1,25%	24,6%
LUTs	6704	9%	2016	2,92%	30,1%
- utilisées comme logique	6294	9%	1976	2,86%	31,4%
- utilisées comme mémoire	410	2%	40	0,06%	9,8%
Bloc RAM	6	4%	0	0,00%	0,0%

Tableau III-XVII: Caractéristiques du watchdog (avec ROM) par rapport au Leon3

	Leon3mp		Watchdog		Watchdog/Leon3mp ratio
	nb	% Virtex5	nb	% Virtex5	
Bascules	3503	5%	890	1,29%	25,4%
LUTs	6704	9%	4418	6,39%	65,9%
- utilisées comme logique	6294	9%	4378	6,33%	69,60%
- utilisées comme mémoire	410	2%	40	0,06%	9,80%
Bloc RAM	6	4%	1	0,68%	16,70%

Bien que l'évaluation soit faite dans des conditions volontairement défavorables, la complexité globale du watchdog reste inférieure à 30% de celle du processeur vérifié. Il faut noter que ce résultat est fortement influencé par les mécanismes qui ont dû être implantés pour gérer le fenêtrage de registres. Ces mécanismes sont très coûteux, ce qui permet d'envisager des coûts

d'implantation du watchdog nettement inférieurs pour des processeurs n'ayant pas une architecture Sparc.

III.7.2. Evaluation des surcoûts mémoire de la méthode IDSM

Pour analyser l'intervalle des surcouts mémoire, nous avons généré le programme du watchdog, pour l'application AES, avec plusieurs seuils de fréquence et de criticité. La taille du programme Leon pour la même application est de 7035 instructions.

Comme indiqué au paragraphe B.1., GCC calcule la fréquence d'exécution de chaque bloc du programme compilé. Cette fréquence est représentée par un réel variant entre 0 et BB_FREQ_BASE. La fréquence du bloc le plus souvent exécuté dans une fonction donnée est initialement fixée à BB_FREQ_BASE et les autres fréquences sont ajustées en conséquence.
Pour chaque bloc linéaire nous calculons avec notre version modifiée de GCC un quotient de fréquences avec l'équation suivante :

$$Frequency\ (BB) = BB\ ->\ frequency * 100 / BB\_FREQ\_BASE$$

Nous avons ensuite choisi 3 seuils de fréquence entre 0% et 101% (0 signifie que toutes les instructions sont fréquentes et 101 signifie qu'il n'y a pas d'instruction fréquente).

La criticité est donnée en unités arbitraires avec une normalisation à deux niveaux. Chaque critère est normalisé à un par rapport à la plus grande valeur calculée. Puis la criticité globale est calculée comme la somme des valeurs obtenues pour chaque critère, divisée par la plus grande des sommes calculées (donc avec le même poids pour tous les critères et toutes les valeurs sont comprises entre 0 et 1). Tout comme la fréquence, nous avons choisi pour la criticité 3 seuils entre 0% et 101% (0% signifie que toutes les variables sont critiques et 101% signifie qu'il n'y a pas de variables critiques).

Tableau III-XVIII : Surcoût mémoire de la méthode IDSM en fonction des seuils de fréquence et de criticité

Seuil de fréquence	Seuil de criticité	Programme Watchdog (Nb d'instructions)	Surcoût mémoire
101%	**101%**	5183	73,67%
	50%	5196	73,86%
	0%	5249	74,61%
50%	**101%**	6080	86,43%
	50%	6081	86,44%
	0%	6101	86,72%
0%	**101%**	6618	94,07%
	50%	6618	94,07%
	0%	6618	94,07%

Comme décrit dans le tableau III-XVIII, les surcoûts mémoire obtenus varient pour l'application AES entre 73% et 94% en fonctions des seuils de criticité et de fréquence sélectionnés.
Cependant, si nous examinons la répartition des instructions dans le pire cas, à savoir la configuration où les seuils de fréquence et de criticité sont à 0%, nous remarquons que nous

avons 1520 instructions (22,96% du programme) dans le programme principal et 5098 instructions en dehors du programme principal dont 1044 instructions de saut (15,77% du programme) et 4054 instructions de non saut (61,25% du programme). Les instructions en dehors du programme principal prennent en réalité 2 et 1 octets pour les instructions de saut et non saut respectivement. Si nous optimisons donc l'organisation de la mémoire, en permettant le stockage de plusieurs instructions par ligne, nous obtenons un surcoût mémoire autour de 46%, surcoût comparable à la méthode WDP (24-61%).

Par ailleurs, l'examen de la répartition des instructions et des tailles réelles qu'elles occupent, présentées dans le tableau III-XIX, nous montre qu'une optimisation de l'organisation de la mémoire, avec la possibilité de stockage de plusieurs instructions par ligne, nous permettrait d'avoir un surcoût mémoire chutant entre 25% et 37%.

Tableau III-XIX: Répartition des instructions watchdog pour l'application AES

Type d'instruction	Taille réelle de l'instruction	Seuil Fréquence 101%		Seuil Fréquence 0%	
		Seuil Criticité 101%	Seuil Criticité 0%	Seuil Criticité 101%	Seuil Criticité 0%
I0	24	32	30	32	30
I1	31	0	0	0	0
I2	32	0	2	0	2
I3	16	0	0	1416	1356
I4	32	0	35	0	37
I5	32	0	29	0	29
I6	28	23	23	23	23
I7	28	14	16	28	28
I8	24	7	7	7	7
I9	20	5	5	5	5
IA	24	533	533	533	533
IB	8	4054	4054	4054	4054
IC	24	198	198	198	198
ID	8	115	115	115	115
IE	24	80	80	80	80
IF	28	122	122	121	121
Taille du programme (en bit)		58304	60424	81324	82492
Surcoût mémoire		25,90%	26,84%	36,12%	36,64%

La figure 3-19 illustre la répartition des types d'instructions dans le programme principal (les autres types étant invariants par rapport aux seuils de fréquences et de criticité, ou très liés à l'architecture du Leon3). Cette figure montre qu'il existe des instructions très peu utilisées quelque soit le seuil de fréquence ou de criticité, à savoir les instructions I1, I2, I8 et I9. Quand les seuils de fréquence et de criticité sont élevés, les types d'instructions les plus récurrents sont I0, I6 et I7. Alors que dans le cas contraire, près de 9 instructions sur 10 sont des instructions I3. Pour réduire les surcoûts en mémoire, il est envisageable de réduire la taille des instructions fréquemment utilisées, comme I0, I3, I6 et I7. Ceci peut être possible si nous adoptons un codage différent, moins régulier, pour le jeu d'instructions du watchdog.

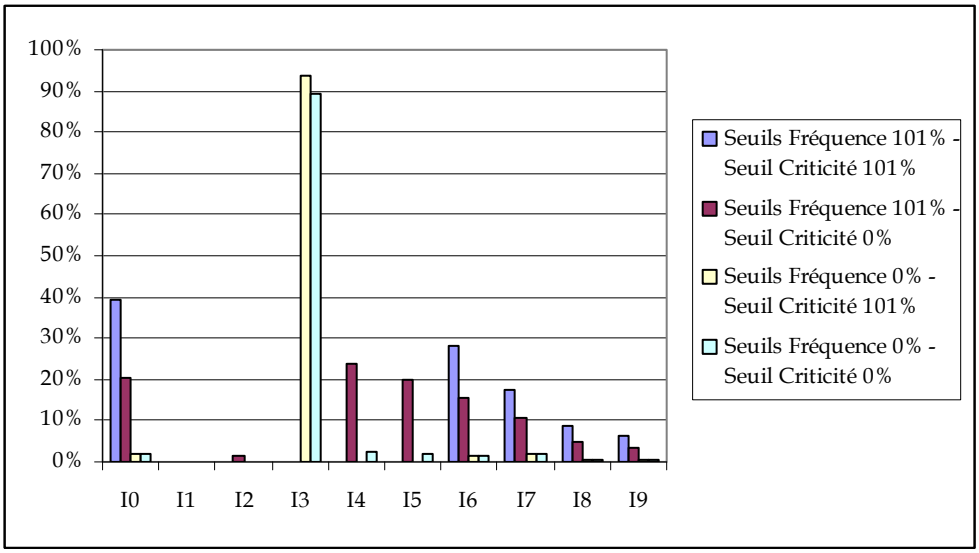


Figure 3-19: Répartition des types d'instruction en fonction des seuils de fréquence et de criticité

III.8. Conclusion

Dans ce chapitre, nous avons cherché à valider par injection de fautes la méthode d'identification des variables critiques. Une première étude sur le 68HC11 a confirmé que les valeurs des critères de criticité correspondent globalement aux effets des injections de fautes et donc à la criticité réelle des variables. Une autre étude sur le Leon2 n'a pas été concluante à cause des mécanismes de gestion des aléas dans le pipeline. D'autres travaux sont en cours, à la fois sur un processeur intermédiaire (le MC68000) et sur une analyse fine du comportement du pipeline pour le Leon3.

Par ailleurs, un prototype de la méthode IDSM a été développé pour évaluer les différents coûts de notre méthode CFC. Ce prototype a été, en plus, soumis à des injections de fautes par émulation pour déterminer l'efficacité de la méthode ainsi que son taux de couverture réel. Les résultats présentés montrent que la méthode IDSM permet de détecter les fautes injectées pendant le boot logiciel (jusqu'à 84% pour les SEU en Fetch-PC) et même dans le boot matériel (jusqu'à 18,5% pour les MBU4 dans le même registre). Par ailleurs, la détection des fautes injectées pendant l'exécution du programme principal dépasse 80% des injections dans le registre Decode-Inst en MBU5 et 90% des injections dans le registre Fetch-PC en SEU.
Ces résultats peuvent être sensiblement améliorés avec l'ajout d'un timer dans le watchdog qui permettrait de détecter les erreurs de délai, considérées dans notre étude comme des cas de non terminaison au bon cycle, tout comme les crashs. De cette manière nous aurons de meilleurs taux de couvertures que les autres méthodes combinées à savoir 30-70% pour [Bens. 03], 24-81% pour [Bens. 01] et 65%-100% pour [Wilk. 97].

Chapitre IV :

Etude des effets des options de compilation sur la criticité des variables

Après avoir proposé la méthode IDSM permettant de détecter les fautes en ligne et donc de réduire l'effet des erreurs sur l'application, nous nous proposons dans ce chapitre d'étudier l'impact des options de compilation sur la criticité des variables d'une application, dans l'optique de trouver la meilleure combinaison d'options de compilation pouvant réduire cette criticité et donc augmenter la robustesse intrinsèque du système.

Dans le paragraphe IV.1 nous préciserons les critères d'évaluation des effets des options de compilation, puis, dans le paragraphe IV.2 nous décrirons la méthodologie adoptée pour notre étude, et enfin, dans les paragraphes IV.3 et IV.4 nous discuterons les résultats obtenus et nous les comparerons avec ceux présentés dans les études de l'état de l'art.

IV.1. Présentation des critères de criticité

Pour évaluer les effets des options de compilation sur la robustesse d'une application, nous étudions l'évolution des critères de criticité suivants :

- Durée de vie (*Lifetime*),
- Participation dans les conditions de branchements (*Weight in the branch conditions*),
- Dépendances fonctionnelles (*Functional dependencies*),
- Nombre de descendants (*Fanout*).

Les trois premiers critères ont été déjà utilisés dans le chapitre II pour l'identification des variables critiques. Il est vrai que nous avons démontré, dans le chapitre III, les limitations du calcul des critères de criticité des registres. Nous pensons toutefois que, notre méthode reste tout à fait utilisable si nous considérons la criticité globale d'une variable sur l'ensemble des registres de la micro-architecture. Dans cette étude, nous nous proposons d'évaluer les critères de criticité de toutes les variables, indépendamment de l'endroit où elles sont stockées. Notre analyse ne doit pas se limiter aux variables stockées en mémoire ou dans le banc de registres mais doit prendre en considération les données temporaires de l'application, stockées dans les registres internes et pouvant être utilisées dans le pipeline.
Dans la suite, nous utiliserons le terme variable pour désigner toutes les variables de l'application mais aussi les données temporaires manipulées pendant l'exécution. En fait, cette définition correspond au terme « pseudo-registre » utilisé dans la documentation des compilateurs.

Il est à noter que la manière de calculer le troisième critère est légèrement différente de celle utilisée précédemment. En effet, pour trouver les dépendances fonctionnelles d'une variable

« v » (Df(v)), nous nous arrêtions à la construction de sa matrice de dépendances, or cette présentation n'étant pas quantifiable, nous utiliserons ici la formule suivante :

$$Df(v) = \sum_{z \subset desc(v)} M(v, z) * (K_l * C_l(z) + K_w * C_w(z))$$

où C_l et C_w sont respectivement la durée de vie et le poids dans les conditions de branchement de la variable « z ». Kl, Kd et Kw sont des coefficients de régulation respectivement associés à la durée de vie, la dépendance fonctionnelle et le poids dans les conditions de branchement.

Le dernier critère, le « fanout », a été communément utilisé dans les méthodes de l'état de l'art pour l'identification des variables critiques d'une application mais aussi pour l'évaluation des effets des options de compilation. Nous n'avons pas utilisé ce critère dans le chapitre II, étant donné que notre formule de calcul de criticité l'englobait dans l'élaboration de la matrice des dépendances fonctionnelles. L'objectif ici n'étant pas d'analyser en détail les variables utilisées pour l'évaluation de la criticité mais de voir globalement l'impact des options de compilation sur la vulnérabilité d'une application, il est intéressant de voir l'évolution de ce critère, seul, et ce en dehors de notre formule de calcul de criticité.

IV.2. Méthodologie d'évaluation des effets des options de compilation

IV.2.1. Analyse statique du code source

L'approche la plus courante pour comparer les effets des options de compilation sur la vulnérabilité d'un logiciel est de réaliser des campagnes d'injections de fautes dans les différentes versions de ce logiciel (chaque version étant compilée avec une option différente), que ce soit par simulation ou par émulation. Malheureusement, cette approche est très coûteuse en termes de temps de développement et dans la plupart des cas ces délais deviennent prohibitifs. Le recours à des tests dynamiques dans les accélérateurs de particules est encore plus coûteux, et ne peut pas être utilisé pour comparer de manière générale plusieurs options de compilation de logiciels.

D'un autre côté, l'analyse statique de code lors de la compilation permet d'obtenir des informations sur la criticité du code généré, à un coût avantageux. Cependant, elle possède des limites intrinsèques pouvant causer des inexactitudes dans les analyses de criticité. Par exemple, l'évaluation de la durée de vie dans les boucles est particulièrement difficile car le nombre exact d'itérations ne peut pas être toujours connu au moment de la compilation, et dans ce cas l'évaluation de la durée de vie des variables de ces blocs peut être biaisée. Le code suivant montre un exemple d'une telle situation, pour lequel il est clair que la durée de vie de X dépend du nombre d'itérations dans la boucle :

```
X = 0;
répéter tant que (condition) { … }
f(X);
```

Cependant, les analyses existant dans les compilateurs modernes seront supposées être assez précises, surtout lorsque nous considérons la moyenne des résultats sur un grand nombre d'exemples de programmes.
L'évaluation des dépendances fonctionnelles d'une variable est également très difficile car de nombreuses instructions dépendent de structures conditionnelles. L'exemple suivant illustre ce cas :

```
X, Y, Z;
f(Y, Z);
if (condition) X = Y;
else X = Z;
```

Ici, la valeur finale de X dépend de « condition » : Si cette condition est vraie, X sera considéré comme un successeur de Y ; dans le cas contraire un successeur de Z. Ce genre de structure a donc un grand impact sur les dépendances fonctionnelles de Y et Z. Etant donné que cette indétermination ne peut être résolue au moment de la compilation, nous considérons que Y et Z sont tous les deux descendants de X.
L'analyse statique échoue également dans le traitement des codes inaccessibles. Parfois, la couverture du code n'est pas parfaite et certaines parties du code source ne doivent jamais être exécutées. Prendre le code inaccessible en compte ou non dans le calcul de criticité des variables peut conduire à des erreurs de jugement, car nous pouvons avoir des évaluations différentes pour des codes qui auraient le même résultat. Cependant, les applications critiques ne contiennent généralement pas de parties inaccessibles, c'est pour cette raison que nous négligerons cet aspect dans notre étude.
En dépit de ces limites, et afin de pouvoir analyser un grand nombre d'exemples de programmes, nous avons donc choisi de nous appuyer sur l'analyse statique que nous avons implanté dans GCC. Le calcul des différents critères de criticité est aussi précis que possible et nous verrons que les résultats obtenus sont cohérents avec des résultats présentés précédemment, notamment ceux basés sur de l'injection de fautes.

IV.2.2. Présentation de l'étude de cas

Dans ce chapitre, nous gardons la même étude de cas utilisée dans le chapitre III et basée sur le processeur Leon3. Le tableau IV-I décrit les principaux registres de cette architecture **[Gais. 02]**. Cette description est utile pour la compréhension des effets des macro-options sur la criticité des registres. Nous avons également gardé, pour quelques expérimentations, les mêmes applications benchmarks que celle du chapitre III à savoir : Mtmx, une application de multiplication de matrices et Fir un filtre à réponse impulsionnelle finie. Outre ces deux programmes, nous utilisons des applications venant de la suite MiBench **[Mibe. 01]** et décrites dans le tableau IV-II.
Nous avons utilisé une version modifiée du compilateur GCC V4.2.4 pour Sparc v8. En effet, les contraintes de l'édition dynamique des liens, décrites dans le chapitre III, n'affectent pas le calcul des critères de criticité ; nous préférons donc utiliser cette version qui offre une analyse du flot de données plus complète que la version 4.4.2.

Tableau IV-I : Les registres Sparc V8

	Registres	Numéro	Description
Global	%g0	0	Toujours égal à 0
	%g1	1	Valeur temporaire
	%g2 -%g4	2-4	Registres globaux
	%g5-%g7	5-7	Réservés pour Sparc ABI
Out	%o0	8	Paramètre sortant (0) / Valeur de retour de la fonction appelée
	%o1-o5	9-13	Paramètres sortants
	%o6	14	Pointeur sur la pile %sp
	%o7	15	Valeur temporaire / adresse de l'instruction CALL
Local	%l0 -%l7	16-23	Registres locaux
In	%i0	24	Paramètre entrant (0) / Valeur de retour de la fonction appelante
	%i1-i5	25-29	Paramètres entrants
	%i6	30	Pointeur sur la fenêtre de registres %fp
	%i7	31	Adresse de retour
	%icc	100	Registre d'état

Tableau IV-II : Présentation des applications MiBench

Benchmark	Description
AES	Fonction standard de chiffrement/déchiffrement
JPEG	Algorithme de compression/décompression d'images
GSM	Standard d'encodage/décodage de la voix.
Basicmath	Cette application effectue des calculs mathématiques simples qui, souvent, n'ont pas de support matériel dédié dans les processeurs embarqués.
Qsort	Le test qsort trie un large éventail de chaînes en ordre croissant en utilisant l'algorithme de tri rapide.
Patricia	Un arbre de Patricia est une structure de données utilisée à la place des arbres complets avec des nœuds feuilles très rares. Patricia est utilisé pour représenter les tables de routage dans les applications réseau.
FFT	Transformée de Fourier rapide sur un tableau de données.
CRC32	Contrôle de redondance cyclique 32-bit sur un fichier.
Blowfish	Chiffrement par blocs symétrique avec une clé de longueur variable.
ADPCM	Algorithme de Codage / décodage, variante de PCM, la modulation par impulsions codées.
Stringsearch	Recherches pour les mots donnés dans les phrases.

Dans la première partie de notre travail, nous avons analysé cinq versions de chaque benchmark, obtenues chacune en changeant seulement la macro-option d'optimisation du compilateur. Le nom et la définition de chaque macro-option sont résumés dans le tableau IV-III **[Gcc 08]**. En plus de ces macro-options, GCC possède de nombreuses fonctions d'optimisation. Ces fonctions sont incluses dans les macro-options O2 et O3. Pour évaluer les effets de ces fonctions nous avons dans un second temps procédé comme suit : chaque compilation a été réalisée avec la macro-option O1 à laquelle est ajoutée individuellement la fonction à évaluer. Le tableau IV-IV présente la liste des fonctions étudiées.

Afin de mesurer l'impact de chaque fonction d'optimisation, nous utilisons l'augmentation de la criticité globale pour chaque critère de criticité, la criticité globale étant la somme de la criticité de toutes les variables. Par exemple, pour évaluer l'effet de l'option *-finline-functions* sur la durée de vie des variables de l'application JPEG, nous suivons les deux étapes suivantes :

1. Nous compilons d'abord JPEG avec -O1 et calculons la somme de toutes les durées de vie obtenues. Appelons cette valeur « base ».
2. Nous compilons ensuite JPEG avec -O1 et l'option *-finline-functions*, et une fois de plus, nous sommons toutes les durées de vie obtenues. Le résultat est par la suite divisé par la valeur de base obtenue lors de la première étape.

Une valeur supérieure à 100% indique une augmentation de la durée de vie totale des variables et donc de la criticité. Par opposition une valeur inférieure à 100% indique une diminution de la criticité et traduit un effet positif de l'option de compilation sur la robustesse.

Tableau IV-III : Les macro-options d'optimisation dans GCC

Macro option d'optimisation	Définition
O0	C'est l'option par défaut. Elle n'effectue aucune optimisation.
O1	Le compilateur essaye de réduire la taille du code et le temps d'exécution sans effectuer les optimisations pouvant augmenter significativement le temps de compilation.
O2	Optimise plus. GCC effectue la majorité des optimisations supportées sauf celles impliquant un compromis taille du programme/vitesse d'exécution. Le compilateur n'exécute pas les options *loop unrolling* et *function inlining*.
O3	Optimise encore plus. '-O3' réalise les options relatives au *function inlining* en plus de toutes les optimisations de O2.
Os	Réduit la taille du programme exécutable. '-Os' utilise toutes les optimisations de '-O2' qui n'augmentent pas la taille du programme.

Tableau IV-IV : Fonctions d'optimisation évaluées

cse-follow-jumps	schedule-insns2	reorder-blocks
cse-skip-blocks	schedule-insns	reorder-functions
delete-null-pointer	strict-overflow	rerun-cse-after-loop
expensive-optimizations	thread-jumps	align-labels
gsce-after-reload	tree-vrp	align-loops
Gcse	tree-pre	crossjumping
gcse-lm	caller-saves	align-jumps
infline-functions	unswitch-loops	regmove
Ivopts	shed-interblock	align-functions
optimize-siblings-calls	shed-spec	peephole2

IV.3. Résultats de l'évaluation de l'impact des options de compilation

Dans ce paragraphe, nous allons dans un premier temps discuter des effets des macro-options d'optimisation, d'abord sur la criticité des registres, ensuite sur la criticité des variables en général quelle que soit leur localisation (registres ou mémoire). Dans un deuxième temps, nous allons augmenter la précision de l'étude pour explorer les effets des fonctions d'optimisation sélectionnées individuellement et ensuite combinées pour certaines d'entre elles. Les résultats détaillés peuvent être trouvés dans l'annexe C.

Les optimisations non supportées par la compilation pour le Sparc v8 ainsi que celles sans effet sur les critères de criticité sont ignorées dans ce qui suit.

IV.3.1. Impact des macro-options sur la criticité des registres

La figure 4-1 présente l'évaluation de la criticité des registres de l'application « Mtmx ». La criticité est donnée en unités arbitraires avec une normalisation à deux niveaux : la valeur de chaque critère, pour un registre donné, est d'abord normalisée par rapport à la plus grande valeur de ce critère pour l'ensemble des registres. Puis la criticité globale est calculée comme la somme des valeurs obtenues pour chaque critère, divisée par 4 (donc avec le même poids pour tous les critères). La valeur exacte calculée n'est pas significative, le but ici est simplement d'analyser l'évolution des critères de criticité en fonction des optimisations de compilation et d'analyser également la contribution de chaque critère par rapport à la criticité globale.

Comme le montre la figure 4-1, la criticité calculée peut atteindre un niveau de plus de 0,6 pour certains registres. Le nombre de registres avec ce niveau de criticité est plus élevé lorsque les optimisations sont activées. Avec l'option O0, 13 registres sont identifiés comme étant plus ou moins critiques; 17 registres ont une certaine criticité avec les optimisations O1 et Os ; et 19

registres sont identifiés avec les options O2 et O3. En fait, pour cet exemple, le niveau d'optimisation O3 ne change pas le résultat par rapport à O2. Toutes les optimisations conduisent donc à une augmentation du nombre de registres critiques. En regardant maintenant critère par critère, nous remarquons que pour l'option O0 la participation dans les conditions de branchement impacte principalement la criticité du registre 100 (%icc). Avec plus d'optimisations, ce critère devient beaucoup plus important pour plusieurs registres. En outre, la durée de vie des registres augmente sensiblement avec les optimisations. Le critère du nombre de descendants « fanout » est moins sensible aux optimisations.

Le tableau IV-V montre la criticité totale évaluée pour chaque critère sur tous les registres utilisés dans l'application Mtmx. Nous remarquons une augmentation significative de la criticité à cause des optimisations, même au premier niveau. Le tableau IV-V montre également la criticité totale évaluée selon chaque critère sur tous les registres pour deux autres benchmarks : FIR et JPEG. Dans le cas de JPEG, une augmentation significative de la criticité des registres est également observée, et la différence entre les niveaux O3 et O2 est visible. Nous pouvons remarquer que l'optimisation Os, dans le cas de Mtmx mène à la pire criticité globale, alors que pour JPEG elle mène à une criticité moyenne. Les résultats obtenus pour JPEG montrent l'incidence négative des optimisations sur les quatre critères, y compris le fanout, alors que FIR montre au contraire une criticité globale qui diminue avec la plupart des optimisations. Ces exemples ont été choisis pour illustrer l'importance de l'application sur l'effet des optimisations.

Tableau IV-V : Criticité globale des registres pour les applications MTMX, FIR et JPEG

		Durée de vie	Fanout	Dépendances fonctionnelles	Participation aux sauts	Total
MTMX	O0	0,80	0,79	1,43	0,58	3,61
	O1	1,61	0,79	1,50	1,25	5,16
	O2	1,83	0,80	1,49	1,25	5,38
	O3	1,83	0,80	1,49	1,25	5,38
	Os	2,07	0,80	1,67	1,25	5,80
FIR	O0	2,40	0,81	1,90	1,25	6,37
	O1	2,39	0,80	1,57	1,25	6,02
	O2	2,51	0,80	1,45	1,25	6,01
	O3	2,18	0,78	1,62	0,79	5,39
	Os	2,40	0,81	1,90	1,25	6,37
JPEG	O0	0,74	1,19	1,94	0,46	4,35
	O1	1,85	1,21	2,17	0,37	5,61
	O2	2,05	1,38	2,39	0,37	6,20
	O3	2,07	1,41	2,91	0,32	6,72
	Os	1,97	1,29	2,23	0,41	5,91

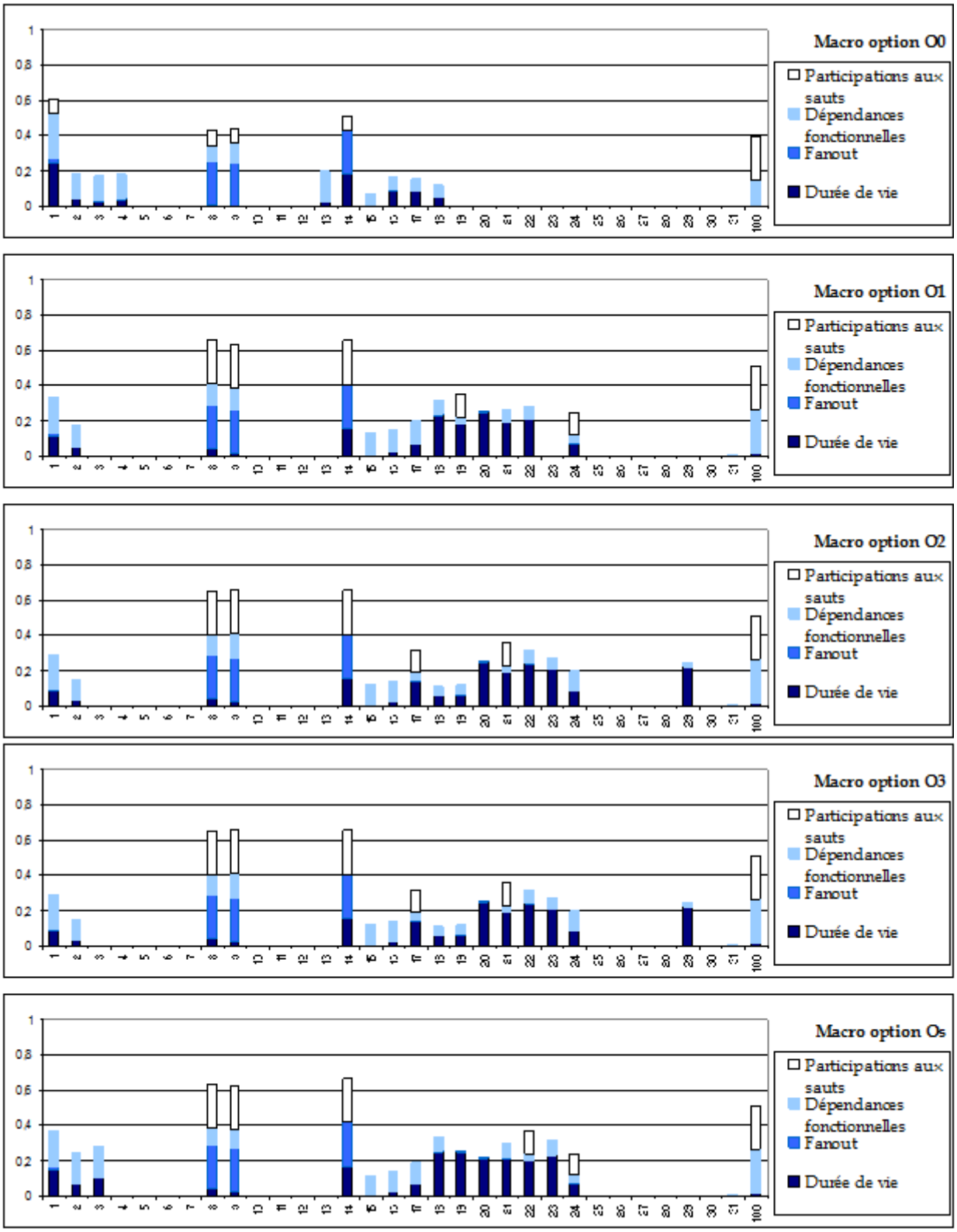


Figure 4-1: Evaluation de la criticité des registres de l'application MTMX

IV.3.2. Impact des macro-options sur la criticité des registres et de la mémoire

IV.3.2.1. Cas de l'application AES

L'application AES, a été choisi pour une étude plus globale que celle basée sur les petits exemples précédents, mais en se focalisant sur le critère de criticité le plus fréquemment employé dans la littérature, à savoir la durée de vie. Le code source disponible a été compilé avec les cinq macro-options et nous avons analysé le temps de compilation, la taille du code généré, le nombre de variables ainsi que la durée de vie totale des variables. Nous avons également analysé la performance des différents programmes obtenus, par simulation avec ModelSim de l'exécution de ces programmes sur le modèle RTL VHDL du processeur Leon3. Tous ces résultats sont résumés dans le tableau IV-VI. Comme prévu, ces résultats montrent qu'il n'y a pas de solution d'optimisation qui améliore globalement la compacité du code et les performances de l'application. Par ailleurs, la criticité est à nouveau augmentée par les optimisations.

Une analyse plus poussée des durées de vie a été réalisée, en séparant les variables en fonction de leur emplacement dans la mémoire ou dans les registres : les figures 4-2 et 4-3 illustrent respectivement la répartition des durées de vie et des nombres de variable, selon leur localisation en mémoire ou en registres internes. Les durées de vie liées aux variables stockées en mémoire pour les options O0 et O1 sont très faibles, voire nulles (toutes les variables sont stockées dans des registres avec O1). Elles sont plus élevées pour les autres options, mais ne dépassent pas 25% de la durée de vie totale. Protéger les mémoires de données ne permet donc pas de réduire significativement la criticité du système.

Tableau IV-VI : Caractéristiques de compilation de l'application AES

	Durée de vie totale	Nombre de variables	Taille du code	Temps d'exécution (ns)	Temps d'exécution (nb cycles)	Temps de compilation (s)
O0	10682	18	3164	1259675	65231	42,81
O1	25750	22	1491	899175	44958	22,27
O2	41740	44	1513	887625	44381	22,7
O3	61178	44	2454	885946	44297	34,02
Os	36378	42	1494	888149	44407	19,71

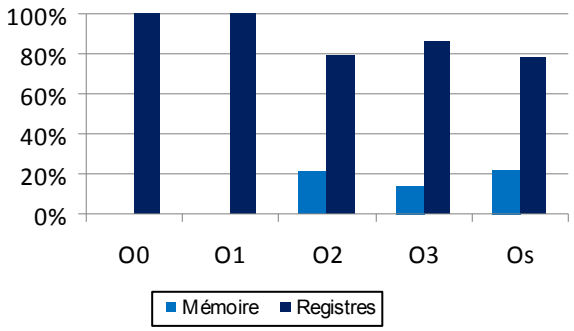


Figure 4-2: Répartition des durées de vie entre mémoire et registres pour l'application AES

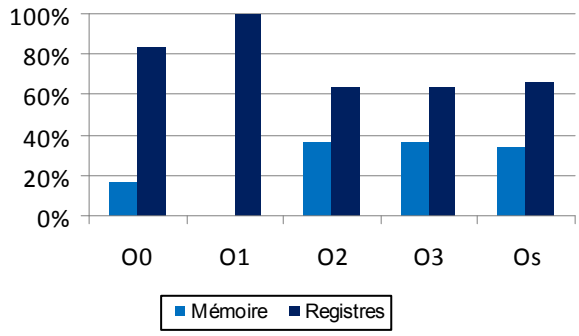


Figure 4-3: Répartition du nombre de variables entre mémoire et registres pour l'application AES

IV.3.2.2. Autres benchmarks

Les résultats détaillés de la compilation des différents benchmarks sont donnés en Annexe C. Nous ne donnons ici que les principaux résultats. La figure 4-4 illustre les variations de la durée de vie totale et la figure 4-5 montre les variations dans le nombre de variables pour chaque macro-option d'optimisation. En raison de la grande variation des résultats entre les différents benchmarks, et afin d'améliorer la lisibilité, les résultats sont normalisés, pour chaque benchmark, par rapport à la valeur la plus élevée obtenue à partir des compilations avec les cinq options. Comme observé dans le paragraphe IV.3.2.1 pour l'AES, la figure 4-5 montre que dans la plupart des cas, les variables sont stockées dans les registres internes. En outre, pour certaines options, la durée de vie des variables en mémoire est négligeable. Le nombre de variables stockées dans les registres est souvent indépendant de l'option de compilation et dans de nombreux cas, représente le principal facteur de la durée de vie globale et donc de la sensibilité absolue aux perturbations.

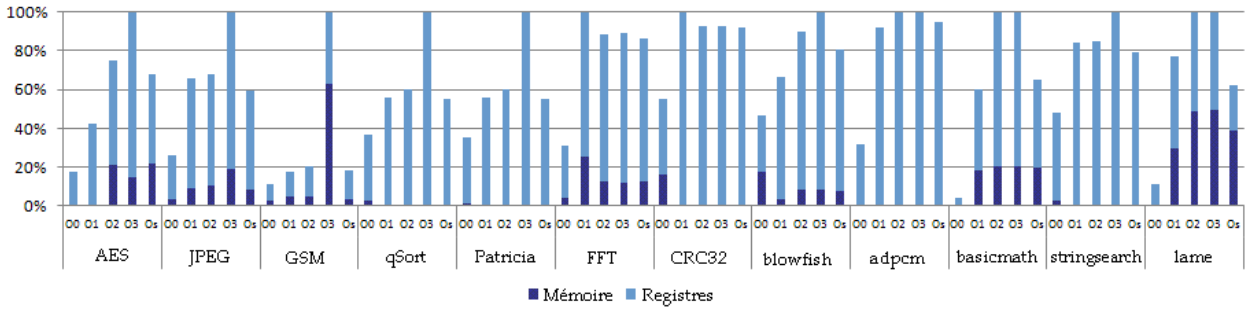


Figure 4-4: Durée de vie totale des variables stockées en mémoire et dans les registres (valeurs normalisées pour chaque benchmark par rapport à la plus grande valeur obtenue pour les cinq macro-options)

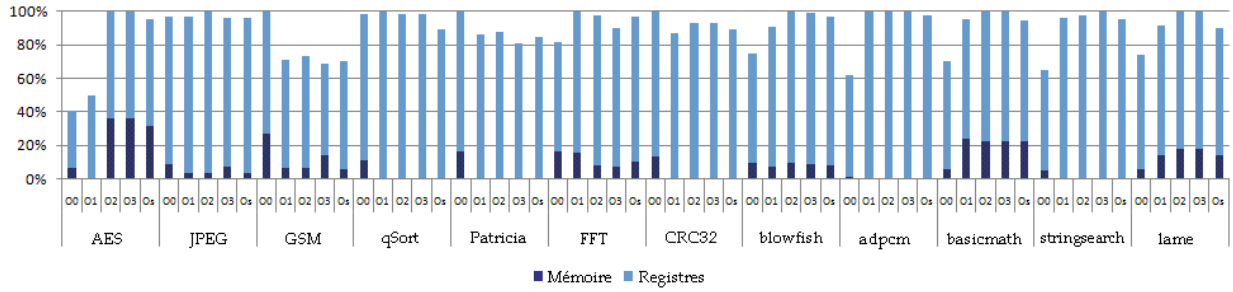


Figure 4-5: Nombre de variables (valeurs normalisées pour chaque benchmark par rapport à la plus grande valeur obtenue pour les cinq macro-options)

Les résultats montrent clairement que le choix de l'option d'optimisation a des répercussions importantes sur les durées de vie des variables que ce soit pour les variables stockées dans des registres ou dans la mémoire. Mais l'impact réel des optimisations sur la durée de vie dépend des caractéristiques du programme d'application. Corréler certaines caractéristiques de l'application avec l'impact d'optimisation n'est pas simple, et n'a pas pu être complètement réalisé pendant cette thèse, mais peut-être un sujet pour les travaux futurs. Deux observations peuvent motiver la poursuite des travaux :

(1) Dans le cas général, les variables stockées dans les registres ont une durée de vie supérieure à celle des variables stockées en mémoire. Cependant, il ya quelques exceptions, comme indiqué dans la figure 4-4, pour les benchmarks GSM et lame, avec quelques options d'optimisation.

(2) La durée de vie ainsi que le nombre de variables varient considérablement en fonction de l'option d'optimisation, mais certaines tendances dépendent de l'exemple :

- Le nombre total de variables ainsi que la durée de vie totale des variables stockées dans les registres sont considérablement plus faibles dans le cas de O0 et représentent même le dixième des valeurs obtenues avec O3 dans le cas de Lame. Le classement des options de compilation en fonction des durées de vie des variables stockées dans des registres est globalement: O0, O1, O2, Os, O3. Naturellement, l'utilisation de O0 a un coût en termes de taille du code et de temps d'exécution. Mais si l'objectif principal est de réduire la sensibilité des registres, O0 reste la meilleure option pour la plupart des exemples analysés. Il reste toutefois le contre-exemple FIR, pour lequel les optimisations sont globalement profitables.

- Pour les variables stockées dans la mémoire, il existe deux familles d'exemples. Pour la première famille (AES, JPEG, GSM, basicmaths, FFT, blowfish et Lame) la meilleure option de compilation est O0 et le nombre de variables stockées dans la mémoire peut augmenter d'un facteur de près de trois, d'une option à l'autre. Pour la deuxième famille (qsort, patricia, CRC32, adpcm et stringsearch) la seule option qui se traduit par une allocation de mémoire est O0. Les caractéristiques de l'exemple ont donc clairement un impact fort sur la criticité des variables en mémoire.

Une première étude sur les caractéristiques intrinsèques des benchmarks a montré que les optimisations permettent de réduire le nombre de variables en mémoire lorsque l'exemple est simple et manipule peu de variables. Elle a également montré que la manière dont le code source a été écrit peut affecter la durée de vie des variables stockées dans la mémoire. Une étude plus approfondie sur la corrélation entre les caractéristiques de l'application et les critères de criticité, sera présentée dans le paragraphe IV.3.3.

IV.3.3. Corrélation entre les caractéristiques de l'application et les critères de criticité

Comme il est mentionné dans le paragraphe IV.3.2, les caractéristiques de l'application peuvent avoir un impact sur l'évolution de la criticité en fonction des options de compilation. Nous nous proposons donc de trouver des corrélations générales. Le tableau IV-VII montre les relations possibles entre les caractéristiques du programme et les critères de criticité pour un sous-ensemble représentatif de benchmarks et pour les quatre macro-options de compilation. Les caractéristiques mentionnées sont la longueur du programme (nombre d'instructions dans le programme principal), la taille moyenne des blocs, le nombre de branchements conditionnels et le nombre de variables. Chaque caractéristique est associée au critère de criticité susceptible d'être le plus affecté par cette caractéristique (dans ce tableau, la référence 100% est calculée pour l'option -O1). Comme le montre le tableau IV-VI, il n'a pas été possible de trouver une corrélation directe entre les critères de criticité et l'évolution des caractéristiques des benchmarks avec les différentes macro-options.

Pour stringsearch, la durée de vie de référence, calculée avec O1, est obtenue pour un programme de 527 instructions. Cette durée de vie a augmenté pour les deux options O2 et O3, alors que O2 diminue le nombre d'instructions et O3 augmente la même caractéristique. La taille moyenne des blocs linéaires est la même avec O2 et O3, mais la durée de vie diffère d'une option à une autre.

Pour AES, le poids dans les conditions de branchement est presque doublé entre O1 et O2, tandis que le nombre d'instructions de branchement a légèrement augmenté. Ce même critère a plus que doublée entre O1 et O3, alors que le nombre d'instructions de branchement a diminué. Pour l'option Os, le poids dans les conditions de branchement a augmenté sensiblement alors que le nombre de branchements conditionnels est resté le même qu'avec O1.

Pour basicmaths, le critère fanout a augmenté de 30% entre O1 et Os, tandis que le nombre de variables (ou «pseudo-registres») a légèrement baissé. D'un autre côté, la valeur de ce même critère a aussi augmenté pour les deux autres options de compilation O2 et O3, mais cette fois-ci avec l'augmentation du nombre de variables.

De nombreux autres exemples peuvent être donnés afin de montrer qu'il n'est pas possible de trouver une tendance générale entre certaines évolutions des caractéristiques de l'application et l'évolution du critère de criticité résultant, au moins pour les caractéristiques que nous avons pris en considération.

Tableau IV-VII: Corrélation entre les caractéristiques de l'application et les critères de criticité

		Macro-option d'optimisation			
		O1	O2	O3	Os
AES	Longueur du programme	1491	1513	2454	1494
	Taille moyenne des blocs	43,85	63,12	117,23	62,25
	Durée de vie globale	100%	153,39%	216,48%	118,52%
	Nombre de branchements conditionnels	22	23	17	22
	Poids dans les conditions de branchement	100%	188,98%	206,37%	176,52%
	Nombre de variables	22	44	44	42
	Fanout	100%	210,43%	226,97%	196,04%
CRC32	Longueur du programme	119	129	129	114
	Taille moyenne des blocs	4,95	4,96	4,96	5,18
	Durée de vie globale	100%	92,81%	92,81%	92,28%
	Nombre de branchements conditionnels	7	7	7	5
	Poids dans les conditions de branchement	100%	110,41%	110,41%	100%
	Nombre de variables	234	237	223	225
	Fanout	100%	113,15%	113,15%	105,26%
Basicmaths	Longueur du programme	746	753	783	719
	Taille moyenne des blocs	7,53	6,78	6,86	7,81
	Durée de vie globale	100%	204,24%	204,24%	109,98%
	Nombre de branchements conditionnels	40	40	44	40
	Poids dans les conditions de branchement	100%	116,94%	116,94%	118,64%
	Nombre de variables	225	237	237	224
	Fanout	100%	121,62%	121,62%	130,27%
qSort	Longueur du programme	90	86	86	68
	Taille moyenne des blocs	5	4,77	4,77	4,53
	Durée de vie globale	100%	111,50%	111,50%	83,05%
	Nombre de branchements conditionnels	8	8	8	6
	Poids dans les conditions de branchement	100%	87,91%	87,91%	82,41%
	Nombre de variables	64	63	63	57
	Fanout	100%	85,33%	85,33%	81,33%
Stringsearch	Longueur du programme	527	520	563	466
	Taille moyenne des blocs	5,43	5,36	5,36	5,23
	Durée de vie globale	100%	101,07%	118,51%	93,77%
	Nombre de branchements conditionnels	58	64	71	52
	Poids dans les conditions de branchement	100%	104,22%	110,25%	94,72%
	Nombre de variables	303	308	316	300
	Fanout	100%	105,82%	110,48%	97,47%

IV.3.4. Impact des fonctions d'optimisation sur la criticité des variables

IV.3.4.1. Fonctions individuelles d'optimisation

Dans cette section, nous nous intéressons aux effets des fonctions d'optimisation, sélectionnées individuellement, sur la criticité des variables.

Comme illustré par la figure 4-6, les fonctions d'optimisation individuelles ont le plus souvent un impact très négligeable sur la durée de vie globale des variables, contrairement à ce que nous avons vu pour les macro-options d'optimisation O2 et O3. Nos expériences montrent que *freorder-functions* et *fdelete-null-pointer-checks* n'ont pas d'impact sur la criticité ; il en est presque de même pour *fregmove* et d'autres fonctions qui semblent avoir un impact très limité. D'un autre côté, nos résultats montrent que les options *fgcse*, *finline-functions, frerun-cse-after-loop*, et *schedule-insns* sont à éviter si l'un des objectifs est d'améliorer la robustesse intrinsèque du système.

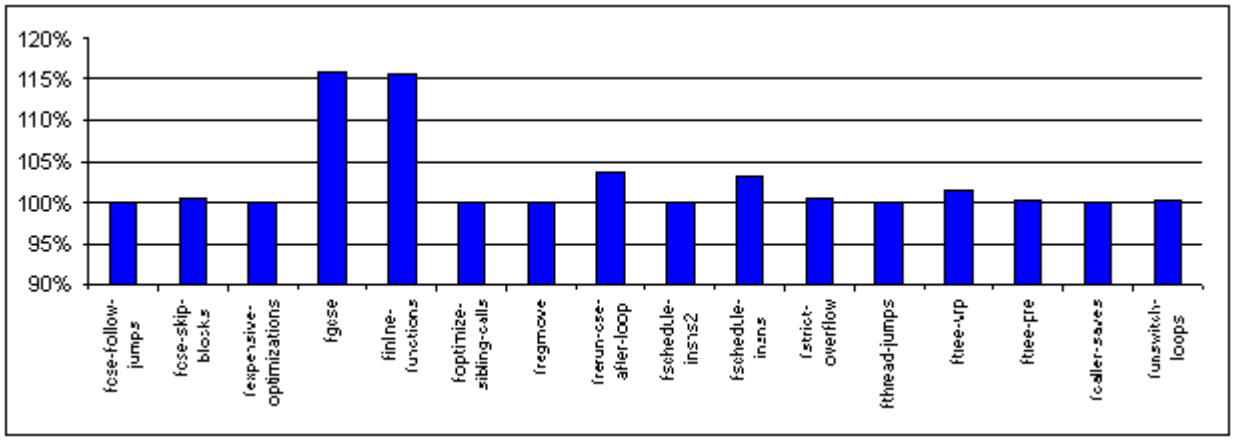


Figure 4-6: Impact moyen sur les durées de vie des fonctions d'optimisation individuelles

La variance de l'impact sur la criticité pour les différentes fonctions d'optimisation est résumée dans le tableau IV-VIII pour les quatre critères de criticité. Certaines fonctions ont une variance très faible, ce qui signifie qu'elles ont le même impact pour les différents benchmarks. Pour *fexpensive-optimizations*, par exemple, la variance est presque nulle : cette option n'a d'effet que sur les exemples GSM et JPEG et cet effet est très limité (une baisse de moins de 1%). Bien que des options telles que *ftree-pre* n'ont pas un taux de variation important (8,63), elles peuvent avoir des effets opposés sur la durée de vie des variables : *ftree-pre* augmente ce critère de 4,26% pour Patricia, alors qu'elle le réduit de 6,3% pour CRC32. En fait, seulement la moitié des options ont le même effet sur tous les benchmarks (une augmentation ou une diminution de la criticité pour tous les exemples), ce qui confirme nos remarques précédentes sur l'importance du code source vis-à-vis des effets des optimisations. *fexpensive-optimisations, fexpensive-optimizations, foptimize-siblings-calls, fregmove, fschedule-insns2* et *fthread-jumps* ont toujours un effet positif sur la durée de vie des variables alors que *fgcse* et *ftree-vrp* ont toujours un effet négatif. Certaines fonctions ont, au moins pour certains critères de criticité, une variance très grande. Parmi les différentes fonctions, la variance atteint 809,1 avec l'option d'élimination des sous-expressions (la fonction *fgcse*) et cela avec une augmentation de la durée de vie des variables qui atteint 96% pour basicmath. Toutefois, les plus grandes valeurs de variances sont obtenues pour les macro-options d'optimisation.

Bien que les valeurs et variances calculées puissent être différentes d'un critère de criticité à un autre, les principales conclusions restent les mêmes. La tendance globale est une diminution de la fiabilité lors de l'utilisation des macro-options d'optimisation, mais aussi pour certaines fonctions individuelles d'optimisation. Les fonctions *fgcse* et *finline-functions*, déjà citées, sont les fonctions qui augmentent le plus la durée de vie des variables (augmentation de 16,1% et 15,7% respectivement). Quelques options peuvent diminuer le niveau de criticité, mais pas de manière significative.

En regardant maintenant les valeurs elles-mêmes, nous remarquons que les gros benchmarks (à savoir GSM ou JPEG qui ont quelques milliers de lignes de code) sont influencés par de nombreuses optimisations, alors que les benchmarks les plus simples (tel que CRC32) ne sont influencés que par de rares optimisations. Cependant, nous n'avons pas remarqué une corrélation entre la taille du code et le niveau de criticité évaluée.

Les données obtenues lors des expériences montrent également que les interactions entre les optimisations ne sont pas faciles à prévoir. Par exemple, la durée de vie des variables dans l'application AES augmente de 41% avec -O3 par rapport à -O2. Pourtant, dans l'ensemble des fonctions individuelles ajoutées entre -O2 et -O3, seule *finline-functions* a un certain effet quand elle est utilisée individuellement et elle n'augmente la criticité que de 5,9%. Un autre exemple frappant concerne le critère de fanout pour l'application qsort. En effet, même si aucune des fonctions d'optimisation ne diminue individuellement ce critère, -O2 et -O3 parviennent quand même à réduire la criticité de 14,7%! La criticité finale obtenue en combinant les fonctions d'optimisation ne peut donc pas être directement déduite des effets de ces fonctions prises individuellement.

Tableau IV-VIII : Variance des critères de criticité par rapport aux fonctions d'optimisation

	Variance			
	Durée de vie	Fanout	Participation aux sauts	Dépendances fonctionnelles
fcse-follow-jumps	0,32	6,65	4,28	4,28
fcse-skip-blocks	0,59	24,66	15,34	15,34
fexpensive-optimizations	0	0	0	0
fgcse	809,16	9,39	4,68	4,68
finline-functions	613,85	31,44	108,12	108,12
foptimize-sibling-calls	0,05	0,038	0,01	0,02
fregmove	0,07	0,063	0,03	0,03
frerun-cse-after-loop	12,9	1,85	1,18	1,18
fschedule-insns2	0,01	55,31	35,47	35,47
fschedule-insns	84,57	765,63	497,92	497,92
fstrict-overflow	1,04	1,75	1,07	1,07
fthread-jumps	0	0	0	0
ftree-vrp	6,98	12,90	8,18	8,18

	Variance			
	Durée de vie	Fanout	Participation aux sauts	Dépendances fonctionnelles
ftree-pre	8,63	3,71	2,11	2,11
fcaller-saves	0	0,11	0,0	0,07
funswitch-loops	0,85	0,11	0,14	0,14
O2	1050,8	1021,4	670,74	670,74
O3	3391,3	1328,4	1351,7	1351,7
Os	124,06	935,62	625,02	625,02

IV.3.4.2. Fonctions combinées d'optimisation

Suite à la remarque précédente, nous avons analysé l'évolution de la durée de vie des variables pour deux types de combinaisons : l'une basée sur les résultats de notre étude et l'autre étant guidée par le manuel de GCC.

Pour la première combinaison, nous avons utilisé les options qui sont toujours bénéfiques, par paires (*fexpensive-optimizations*, *foptimize-sibling-calls*, *fregmove* et *fthread-jump*). Malheureusement, cela ne conduit pas à une diminution significative de la criticité, avec une baisse moyenne de 0,15%. Nous avons ensuite combiné les pires fonctions (*fgcse* et *finline-function*). Cela augmente la criticité moyenne de 31,9% avec un maximum de 76,69% dans le cas de l'application Patricia.

Le deuxième ensemble de combinaisons est basé sur les optimisations elles-mêmes. L'effet de certaines fonctions peut être étendu par l'ajout d'autres drapeaux mineurs. Nous avons en particulier évalué les effets de l'élimination des sous-expressions (*fgcse*), combinée avec *fgcse-fm, fweb* ou *finline-function*. Une variation importante a été observée, mais l'impact sur le niveau de criticité a été négatif dans tous les cas et pour tous les benchmarks (sauf pour CRC32 où aucun impact n'a été enregistré).

IV.4. Comparaison avec les résultats antérieurs publiés

Quelques travaux antérieurs ont étudié l'impact des optimisations du compilateur afin de déterminer la relation entre les optimisations logicielles et les défaillances. **[Pero. 08]** a mis en évidence les variations dans le nombre d'erreurs et les délais, en fonction des options de compilation, après injection de fautes. **[Jone. 08]** a tenté de trouver des combinaisons de fonctions d'optimisation afin d'obtenir le meilleur compromis entre le temps d'exécution et le facteur de vulnérabilité architecturale (AVF), l'AVF étant défini comme la probabilité qu'un défaut dans la structure se traduit par une erreur visible du système **[Mukh. 03].**

Tous les travaux s'accordent sur le fait qu'il est impossible d'identifier une option d'optimisation qui offre toujours le meilleur compromis entre la compacité du code, le temps d'exécution et la robustesse. En outre, la conclusion était qu'en moyenne, la meilleure option est l'option O0, ce qui signifie qu'éviter les optimisations est globalement le meilleur choix en ce qui concerne la robustesse. Bien sûr, un tel choix est coûteux en termes de performances et /ou taille de la

mémoire instructions. Les études antérieures avaient été limitées par les durées expérimentales liées aux injections de fautes, ou bien s'étaient concentrées sur l'AVF. Nous avons cherché à aller plus loin, avec une analyse plus complète en termes de critères de criticité et de nombre d'exemples de programmes analysés. De plus, nous avons cherché à séparer les effets des optimisations sur l'impact éventuel de la protection de la mémoire de données. Nos résultats sont cohérents avec les conclusions antérieures, mais permettent de mettre en évidence plusieurs points qui n'avaient pas été précédemment indiqués, comme l'impact potentiellement faible de la protection des mémoires de données ou l'impact plus particulièrement négatif de certaines fonctions d'optimisation sur la robustesse intrinsèque.

IV.5. Conclusion

Le choix des options d'optimisation lors de la compilation peut avoir un fort impact sur la sûreté de fonctionnement, quel que soit le critère de criticité utilisé. Malheureusement, nous n'avons pas trouvé de fonction d'optimisation ayant un impact positif pour tous les benchmarks et pour tous les critères. Un impact positif peut être observé pour certaines fonctions appliquées à certains programmes, mais en général, l'amélioration reste très faible. A l'inverse, l'utilisation de certaines autres optimisations peut induire une augmentation significative de la criticité. Les fonctions telles que *fgcse* et *finline-functions* doivent être utilisées avec précaution. En outre, les macro-optimisations peuvent conduire à une augmentation de la criticité dans la plupart des cas, quel que soit le critère de criticité évalué. Il est donc nécessaire de faire des compromis entre la robustesse et le temps d'exécution, la taille du code et donc de la mémoire instructions, ou la consommation d'énergie.

Conclusion générale et perspectives

Au cours de cette thèse, une nouvelle méthode de vérification de flot de contrôle a été conçue et implantée. Cette méthode, IDSM, est une méthode à signatures disjointes qui se distingue par rapport aux approches existantes par ses extensions flexibles et configurables par l'utilisateur. En effet, la méthode IDSM permet de faire un monitoring continu de l'exécution. Seulement, par soucis d'économie en mémoire cela n'est fait que pour les instructions fréquemment exécutées. Bien évidement, le seuil de fréquence est laissé au choix de l'utilisateur, qui pourra l'augmenter et le diminuer à sa guise en fonction de ses objectifs : diminution de la latence ou du surcoût mémoire ou un compromis entre les deux. La deuxième extension proposée dans IDSM permet de vérifier les variables critiques. Pour cela, un nouvel algorithme pour le calcul de la criticité à été élaboré, prenant en compte les durées de vies, les dépendances fonctionnelles et la participation dans les conditions de branchement. La méthode IDSM permet de détecter tous les types d'erreurs de flot de contrôle, sauf la correction des chemins pour des raisons de coût, et permet aussi de vérifier l'intégrité des données critiques et signaler leur corruption. De nombreuses contraintes liées à l'architecture du processeur et au programme vérifié ont été prises en compte.

Les outils de développement nécessaires ont été implantés. Le compilateur GCC a été étendu pour permettre la génération automatique du programme exécuté par le watchdog. Des outils complémentaires ont été développés pour gérer notamment les problèmes liés aux calculs d'adresses en phase d'édition des liens. Un prototype matériel a été développé, basé sur le processeur Leon3. Ce prototype a été validé avec des campagnes d'injections de fautes qui nous ont permis de quantifier le taux de couverture des erreurs pouvant survenir dans différents éléments du système. Des taux significatifs de détection ont été obtenus pour des erreurs multiples, difficiles à détecter avec d'autres méthodes. Ce travail a aussi mis en évidence des limitations au niveau du processeur Leon3, et l'intérêt d'ajouter une fonction timer.

Globalement, la première contribution de cette thèse est donc la démonstration de la faisabilité d'une vérification de flot de contrôle avec signatures disjointes, même dans le cas de processeurs modernes avec des architectures exploitant de nombreux mécanismes entravant cette vérification. Cette vérification permet la détection d'un très fort pourcentage d'erreurs de multiplicité élevée, qui ne seraient pas détectées avec des approches plus classiques.

Par ailleurs, une étude sur l'amélioration de la robustesse des applications logicielles a été également effectuée. En effet, disposant des critères de criticité et des outils nécessaires pour les calculer, nous avons pu évaluer les effets des options de compilation sur la criticité des variables. Bien que nous ayons aussi montré par des injections de fautes les limitations de ces critères pour des architectures pipeline, notamment en ce qui concerne les durées de vie dans le banc de registres, nous pensons qu'ils restent tout à fait utilisables en considérant la durée de vie globale d'une information sur l'ensemble des registres de la micro-architecture. Des résultats significatifs ont été obtenus sur ce sujet et ont fait l'objet de plusieurs publications.

De nombreuses perspectives peuvent être citées suite à ces travaux.

A court terme, le prototype peut être étendu pour inclure la vérification des variables critiques et la prise en compte des interruptions. L'organisation de la mémoire du watchdog pourrait aussi être optimisée afin de réduire les surcoûts en mémoire. Ceci nécessiterait l'ajout d'un module de pré-fetch au watchdog et augmenterait sa complexité. Les surcoûts en mémoire pourraient aussi être diminués en adoptant un codage différent des instructions du watchdog, moins régulier, en tenant compte de la fréquence d'utilisation des différents types d'instructions pour réduire le nombre de bits des instructions les plus employées. Il est aussi possible d'imaginer un codage différent visant, sur la même base, à augmenter le nombre de bits vérifiés sur les adresses ou les codes d'instructions, afin d'augmenter le taux de couverture.

La détection des erreurs dans le contenu des instructions en dehors du programme principal peut aussi être améliorée. Pour cela deux solutions sont possibles : soit nous gardons le mécanisme de calcul et de comparaison des signatures horizontales mais en ciblant les 4 bits contrôlés sur les bits les plus critiques, soit nous ajoutons au watchdog un nouveau module de calcul de signatures contrôlant tous les bits de l'instruction.

Le nombre de faux positifs pourrait aussi être réduit en ne tenant compte que des bits effectivement utilisés dans les différents types d'instructions. Ceci augmenterait la complexité du watchdog, en demandant l'ajout d'un masquage dynamique de certains bits lors de la génération de signature, mais pourrait permettre de mieux répondre aux besoins d'applications ayant de fortes contraintes de disponibilité.

Les résultats d'injection de fautes obtenus peuvent être raffinés pour distinguer les délais et retards d'exécution. Nous pourrons par la suite ajouter à notre watchdog un timer pour détecter les erreurs de délais mais aussi certains dysfonctionnements du processeur.

L'évaluation de la criticité des registres est également un point pouvant être davantage développé. L'étude sur le processeur Leon2 a mis en évidence l'impact des mécanismes d'optimisation du pipeline sur la criticité réelle des différents registres. Un objectif à moyen terme est de pouvoir affiner les évaluations de criticité en modélisant le comportement de ces mécanismes. Cet aspect n'a pas été présenté dans ce manuscrit, mais une étude de ce sujet est en cours d'élaboration.

A plus long terme, l'application de la méthode à d'autres processeurs (comme les processeurs ARM, quasiment omniprésents dans les systèmes embarqués) pourrait permettre d'envisager une valorisation industrielle.

Enfin, concernant notre étude des effets des options de compilation sur la robustesse d'une application, une perspective ambitieuse serait la définition de nouvelles options de compilation axées sur la robustesse et prenant ce critère en compte au même titre que les critères classiques de vitesse de calcul ou de coût mémoire.

Bibliographie

[Alkh. 99] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection", IEEE Transactions on Parallel and Distributed Systems, Vol.10, NO.6, June 1999

[Angh. 00] L. Anghel, "Les Limites Technologiques du Silicium et Tolérance aux Fautes", Thèse de Doctorat, Laboratoire TIMA, INPG Grenoble, Décembre 2000.

[Azam. 10] J. Azambuja, M. Altieri, F. Kastensmidt, M. Hübner, J. Becker, "Non intrusive Hybrid Signature-Based Technique to Detect SEU and SET Faults in Microprocessors", 11th European Conference on Radiation and Its Effects on Components and Systems (RADECS2010), Septembre 2010, .

[Azam. 11] J. Azambuja, A. Lapolli, L. Rosa, F. Kastensmidt, "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique", IEEE Transactions on Nuclear Science, Vol. 58, pp. 993 – 1000, June 2011.

[BarE. 06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, "The sorcerer's apprentice guide to fault attacks", Proceedings of the IEEE, vol. 94, no. 2, February 2006, pp. 370-382.

[Baum. 01] R. Baumann, "Soft Errors in Advanced Semiconductor Devices – Part 1 : The Three Radiation Sources", IEEE Trans. on Device and Materials Reliability, vol. 1, issue 1, pp. 17-22, Mars 2001.

[Bens. 00] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, "A C/C++ Source to Source Compiler for Dependable Applications", IEEE Asian Test Symposium (ATS 2001), Kyoto (J), November 2001, pp.209-303

[Bens. 01] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, "Control-Flow Checking Via Regular Expressions", IEEE Asian Test Symposium (ATS 2001), Kyoto (J), November 2001, pp.209-303

[Bens. 03] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "A Watchdog Processor to Detect Data and Control Flow Errors", 9th IEEE International On-Line Testing symposium, Kos, Greece, July 7-9, 2003, pp. 144-148

[Bern. 05] P. Bernardi, L.Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante, "On-line detection of control flow errors in SoCs by means of an infrastructure IPcore", 2005 International Conference on Dependable Systems and Networks (DSN), 2005, pp. 50-58

[Bern. 06] P. Bernardi, L.Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante, "A New Hybrid Fault Detection Technique for systems-on-a-chip", IEEE transactions on Computers, vol. 55, n°2, February 2006, pp. 185-198

[Bori. 06] E. Borin, C. Wang, Youfeng Wu, Guido Araujo, "Software-Based Transparent and Comprehensive Control-Flow Error Detection", International Symposium Code Generation and Optimization, March2006, pp. 333-345. Digital Object Identifier 10.1109/CGO.2006.33

[Chen 05] Y. Chen, "Concurrent Detection of Control Flow Errors by Hybrid Signature Monitoring ", IEEE transactions on Computers, vol. 54, no.10, October 2005

[Cour. 91] B. Courtois, M. Gaudel, J. C. Laprie, D. Powell, "Sûreté de fonctionnement informatique Evolutions 1987-1992 - tendances et perspectives", Rapport de recherche - I.M.A.G 905-I (ISSN), IMAG-RR--92-905-I, LAAS--92-382, LRI--92-786.

[Gcc 08] Free Software Foundation. GCC, the GNU compiler collection. GCC.gnu.org

[Gais. 02] J. Gaisler, "A portable and fault-tolerant microprocessor based on the Sparc v8 architecture", IEEE International Conference on Dependable Systems and Networks (DSN), pp. 409-415, 2002.

[Golo. 03] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, “Soft-error detection using control flow assertions”, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003 pp. 581-588

[Jone. 08] T. M. Jones, M.F. P O’Boyle, O. Ergin, "Evaluating the Effects of Compiler Optimisations on AVF", Workshop on Interaction Between Compilers and Computer Architecture (INTERACT-12), 2008

[Lapr. 85] J.C. Laprie, “Sûreté de fonctionnement des systèmes informatiques et tolérance aux fautes”, Tech. Et Sciences Informatiques, 4 (5), pp. 419-429 Sept-Oct 1985

[Lapr. 04] J.-C. Laprie, "Sûreté de fonctionnement informatique: concepts, défis, directions", ACI Sécurité et Informatique, Toulouse, Novembre 2004.

[Lee 09] J. Lee, A. Shrivastava, “Static analysis to mitigate soft errors in register files“, Design, Automation and Test in Europe Conference (DATE), pp. 1367-1372, 2009.

[Lee 10] J. Lee, A. Shrivastava, “A Compiler-Microarchitecture Hybrid Approachto Soft Error Reduction for Register Files”, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 29, NO. 7, pp. 1018-1027, JULY 2010

[Lee 11] J. Lee, A. Shrivastava, “Static Analysis of Register File Vulnerability”, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 30, NO. 4, pp. 607- 616, APRIL 2011.

[Leek. 10] M. Leeke, A. Jhumka, ”Towards Understanding the Importance of Variables in Dependable Software”, Dependable Computing Conference (EDCC2010), pp. 85 – 94, 2010

[Leek. 11] M. Leeke, A. Jhumka, ”An Automated Wrapper-based Approach to the Design of Dependable Software”, 4th International Conference on Dependability (DEPEND'11), August 21-27th 2011, Nice, France.

[Leve. 90] R. Leveugle, “Analyse de signature et test en ligne intégré sur silicium ”, Thèse en microélectronique INPG, Grenoble, Janvier 1990

[Leve. 09] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, “Statistical Fault Injection: Quantified Error and Confidence“, Design, Automation and Test in Europe, Date 2009, pp 502-506

[Mcfe. 95] L. Mcfearin and V.S.S. Nair, “Control-Flow Checking Using Assertions”, Proc. IFIP Int’l Working Conf. Dependable Computing for Critical Applications, Sept. 1995

[Made. 91] H. Madeira, J. Silva, “On-Line testing: Learning and Checking”, 2nd International Working Conf. on Dependable comp. for Critical Applications, Tucson, 18-20 February 1991, pp. 170-177.

[Mari. 06] R. Mariani, G. Boschi, F. Colucci, “Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508”, Design, Automation and Test in Europe Conference (DATE), April 16-20, 2007, pp. 492-497

[Mari. 07] R. Mariani, P. Fuhrmann, B. Vittorelli, ”Fault-robust microcontrollers for automotive applications”, 12th IEEE International On-Line Testing symposium, Como, Italy, July 10-12, 2006, pp. 213-218

[Mibe. 01] M. Guthaus et al., "Mibench: A free, commercially representative embedded benchmark suite", IWWC, 2001

[Mich. 91] T. Michel, R. Leveugle, G. Saucier, “A New Approach to Control Flow Checking Without Program Modification”, 21th International Symposium on Fault-Tolerant Computing (FTCS-21), pp. 334-341, 1991.

[Mich. 93] T. Michel, “Test en ligne des systèmes à base de microprocesseur ”, Thèse en microélectronique INPG, Grenoble, Mars 1993

[Mich. 94] T. Michel, R. Leveugle, G. Saucier, R. Doucet, P.Chapier, “Taking Advantage of ASICs to Improve Dependability with very Low Overheads”, European Design and Test Conference, 1994.

[Mukh. 03] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003, pp.29-40.

[Namj. 82] M. Namjoo, “Techniques for concurrent testing of VLSI processor operation”, International Test Conference (ITC), 1982, pp.461-468.

[Oh 02] N. Oh, P. P. Shirvani, E.J. McCluskey, “Control Flow Checking by Software Signature”, IEEE transactions on Reliability, vol. 51, no. 2, March 2002, pp. 111-122

[Patt. 09] K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer, “Application-Based Metrics for Strategic Placement of Detectors”, Pacific Rim Dependable Computing (PRDC), 2005

[Pero. 08] P. Peronnard, R. Velazco, G. Foucard, "Impact of the software optimization on the Soft Error Rate: a case study", 23rd International Conference on Design of Circuits and Integrated Systems (DCIS'08), 2008

[Schm. 82] M. Schmid, R. Trapp, A. Davidoff and G. Masson, “Upset Exposure by means of Abstraction Verification ”, Proc. 12th IEEE Fault-Tolerant Computing Symp. , pp 237-244, 1982

[Vanh. 06] P. Vanhauwaert, R. Leveugle, P. Roche, “A flexible SoPC-based fault injection environment”, 9th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS), pp. 192-197, 2006

[Vanh. 08] P. Vanhauwaert, “Analyse de sûreté par injection de fautes dans un environnement de prototypage à base de FPGA “, Thèse de Doctorat, Laboratoire TIMA, IP Grenoble, Avril 2008.

[Vemu 06] R. Vemu, J. A. Abraham, “CEDA: control-flow error detection through assertions”, 12th IEEE International On-Line Testing symposium, Como, Italy, July 10-12, 2006, pp. 151-156

[Vemu 08] R. Vemu, J. A. Abraham, “Budget-dependent control-flow error detection”, 14th IEEE International On-Line Testing symposium, Rhodes, Greece, July 7-9, 2008, pp. 73-78

[Wilk. 88] K. Wilken, J.P.Shen, “Continuous signature monitoring: efficient concurrent-detection of processor control errors ”, International Test Conference (ITC), 1988, pp. 914-925

[Wilk. 90] K. Wilken, J.P.Shen, “Continuous signature monitoring: Low-Cost Concurrent Detection of Processor Control Errors ”, IEEE transactions on Computer-Aided Design, vol. 9, no.6, June 1990

[Wilk. 93] K. Wilken, T. Kong, “Efficient Memory Access Checking”,Fault-Tolerant Computing, 1993. FTCS-23, pp. 566-575

[Wilk. 97] K. Wilken, T. Kong, “Concurrent Detection of Software and Hardware Data-Access Faults”, IEEE Transactions on Computers, vol. 46, no. 4, April 1997

[Yau 80] S. S. Yau, F.-C. Chen, “An approach to concurrent control flow checking“, IEEE transactions on Software Engineering, vol. SE-6, no. 2, March 1980, pp. 126-137

[Zara. 10] H. Zarandi, M. Maghsoudloo, N. Khoshavi, “Two Efficient Software Techniques to Detect and Correct Control-Flow Errors”, IEEE 16th Pacific Rim International Symposium (PRDC2010), 13-15 Dec. 2010, pp. 141 - 148

Publications de l'auteure

Revues internationales :

S. Bergaoui, P. Vanhauwaert, R. Leveugle, "A new critical variable analysis in processor-based systems", IEEE Transactions on Nuclear Science (TNS), (ISSN: 0018-9499), Vol 57, #4, pp 1992–1999, August. 2010

S. Bergaoui, A. Wecxsteen, R. Leveugle, " Detailed Analysis of Compilation Options for Robust Software-based Embedded Systems ", Journal of Electronic Testing: Theory and Applications (JETTA) , (ISSN: 0923-8174), April 2013

Conférences internationales :

S. Bergaoui, P. Vanhauwaert, R. Leveugle, 'A new critical variable analysis in processor-based systems', 10th European Conference on Radiation Effects on Components and Systems (RADECS), Bruges, Belgium, September 14-18, 2009

S. Bergaoui, R. Leveugle, 'IDSM: An improved control flow checking approach with disjoint signature monitoring', Conference on Design of Circuits and Integrated Systems (DCIS), Zaragoza, Spain, November 18-20, 2009

S. Bergaoui, R. Leveugle, "Impact of compilation options on the criticality of registers in a microprocessor-based system", 1st IEEE Latin American Symposium on Circuits and Systems (LASCAS), Iguaçu Falls, Brazil, February 24-26, 2010, pp. 216-219

S. Bergaoui, R. Leveugle, "Impact of Software Optimization on Variable Lifetimes in a Microprocessor-Based System", IEEE 6th International Symposium on Electronic Design, Test and Application (DELTA), Queenstown, New Zealand, January 17-19, 2011

A. Wecxsteen, S. Bergaoui, R. Leveugle, "Detailed Analysis of Compilation Options for Robust Software-based Embedded Systems", 13th IEEE Latin American Test Workshop (LATW), Quito, Ecuador, April 11-13, 2012

Conférences nationales :

S. Bergaoui, R. Leveugle, "Nouvelle Méthode de Vérification de Flot de Contrôle avec Signatures Disjointes", Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM), Montpellier, France, Juin 7-9, 2010

Glossaire

C

CEDA	Control-flow Error Detection through Assertions
CFC	Control flow checking
CFCSS	Control Flow Checking by Software Signature
CSM	Continuous Signature Monitoring

D

DSS	Data Structure Signature
DSM	Disjoint Signature Monitoring

E

ECCA	Emproved Control-Flow Checking Using Assertions
ESM	Embedded Signature Monitoring

I

IDSM	Improved Disjoint Signature Monitoring

L

LFSR	Linear Feedback Shift Register

M

MCV	Most Critical Variables : les variables dont la valeur peut modifier le flot de contrôle de l'application et donc aboutir à une exécution légale mais en produisant des résultats erronés
MISR	Multiple Input Shift Register
MMU	Memory Management Unit : unité qui permet une gestion avancé de la mémoire, avec notamment la pagination, indispensable pour la plupart des OS modernes comme pour exemple Linux

O

OSLC	On-Line testing: Learning and Checking

P

Pipeline	Architecture qui découpe l'exécution d'une tache en plusieurs étages qui sont opéré en parallèle, chacun sur une instruction différente.

R

RECCO	Reliable code compiler

S

SIHFT	Software implemented Hardware Fault Tolerance

Annexe -A-

Présentation du jeu d'instructions du watchdog

Le jeu d'instruction que nous allons présenter dans cette annexe est un jeu d'instructions spécifique à l'architecture Sparc v8 adoptée pour notre prototype. Voici en rappel les instructions de base de notre jeu d'instruction présentées dans le paragraphe II.4:

- I_0 : Destination
- I_1 : Destination load
- I_2 : Destination store
- I_3 : Instruction fréquente
- I_4 : Instruction load
- I_5 : Instruction store
- I_6 : Branchement conditionnel
- I_7 : Branchement inconditionnel
- I_8 : Branchement vers un sous programme
- I_9 : Retour d'un sous programme
- I_{10}* : Instruction de saut en dehors du programme principal
- I_{11}* : Instruction de non saut en dehors du programme principal
- I_{12}* : Branchement vers un sous programme en dehors du programme principal
- I_{13}* : Retour d'un sous programme en dehors du programme principal

A ces instructions, nous ajoutons deux nouvelles instructions définies pour contourner le problème de fenêtrage de registres décrit dans le paragraphe II.2

- I_{14}* : Instruction save
- I_{15}* : Instruction restore

A.1. Présentation des informations nécessaires à chaque instruction

A partir de la description du comportement du watchdog, nous pouvons établir une liste exhaustive des champs nécessaires pour le programme du watchdog. Ces derniers sont présentés dans le tableau A-I.

Tableau A-I: Champs nécessaires au jeu d'instructions du watchdog

Champ	Description
Type	Ce champ informe le watchdog sur le type de singularité rencontrée.
Signature horizontale (SH)	La signature horizontale est obtenue grâce à la compaction de l'instruction en cours d'exécution avec la signature intermédiaire. Elle permet de réduire

	le temps de latence de détection. Remarque : pour les instructions générées après l'édition des liens, les signatures horizontales calculées ne dépendront que de l'instruction en cours (Signature intermédiaire nulle).
Signature verticale (SV)	La signature verticale permet non seulement de vérifier qu'il n'y a pas eu de sauts illégaux mais aussi de vérifier que les instructions exécutées n'ont pas été corrompues. Elle est obtenue grâce à la compaction des instructions du bloc en cours d'exécution.
Adresse de la variable dans la mémoire du watchdog (@V)	Les variables critiques du programme principal vont être stockées dans la mémoire du watchdog pour que ce dernier puisse vérifier leur intégrité à chaque opération les impliquant.
Taille	Ce champ représente le nombre d'instructions du bloc en cours d'exécution. Il doit impérativement être présent dans les nœuds annonçant les débuts des blocs.
Adresse de l'instruction dans le programme du processeur (@P)	Quand le watchdog arrive sur un nœud de destination il doit pouvoir vérifier que le processeur a effectué un branchement vers la bonne instruction. Pour cela chaque nœud de destination doit contenir l'adresse de l'instruction correspondante dans la mémoire du microprocesseur. Ce champ est calculé comme suit. Pendant la compilation et donc bien avant l'édition des liens et la connaissance des adresses, ce champ est calculé comme étant une adresse relative par rapport au début du « main ». De ce fait, ce champ peut être positif ou négatif. Pendant la deuxième phase de génération du programme du watchdog, l'adresse du début du « main » est stockée. Une simple opération d'addition permet de retrouver l'adresse absolue de l'instruction dans le programme principal (une partie de l'adresse absolue pour être plus exact).

Adresse de l'instruction de destination dans le programme du watchdog (@W)	Grâce à ce champ, le watchdog est en mesure de charger le nœud de destination, quand il rencontre une instruction de branchement. Une fois ce nœud chargé, le watchdog compare l'adresse du nœud chargé avec celle du nœud pris par le processeur.
Adresse relative de l'instruction dans la mémoire du processeur (@R)	Ce champ permet de repérer les instructions de types I_3, I_4 et I_5 par rapport à la fin du bloc. Par conséquent, la valeur de ce champ correspond à la valeur du décompteur qui est mis à jour à chaque instruction processeur, à partir de la taille du bloc (retrouvée en début de bloc), et permet d'identifier les fins de blocs.
Flag1	Ce flag indique si l'instruction relative au branchement retardé va être annulée ou pas. Il permet au watchdog de savoir s'il faut qu'il compacte l'instruction qui suit un saut.
Flag2	Ce flag indique si l'instruction restore est également une instruction destination.

A.1.1. Tailles minimales des champs

Les tailles minimales nécessaires pour les champs du jeu d'instructions sont représentées dans le tableau A-II:

Tableau A-II: Champs nécessaires au jeu d'instructions du watchdog

Champ	Description	Taille
Type	Le champ type peut être représenté sur 4 bits puisque notre jeu d'instructions ne comporte que 16 types d'instructions.	4
Signature horizontale (SH)	4 bits peuvent suffire pour vérifier l'intégrité de l'instruction avec une bonne probabilité.	4
Signature verticale (SV)	16 bits peuvent suffire pour représenter cette signature.	16

Adresse de la variable dans la mémoire du watchdog (@V)	Les adresses des variables seront représentées sur 8 bits ce qui permettra au watchdog de surveiller jusqu'à 256 variables.	8
Taille	Etant donné que la taille moyenne d'un bloc ne dépasse pas 10 instructions, 4 bits sont suffisants pour coder ce champ. Dans le cas où la taille du bloc dépasserait 15 instructions, on propose le découpage de ce bloc, de telle sorte que la 15ème instruction soit une instruction de saut inconditionnel vers l'instruction suivante. Mais étant donné que ce découpage peut être coûteux en mémoire instruction (2 instructions watchdog ajoutées pour chaque bloc qui dépasse 15 instructions), la taille de ce champ peut être sur un octet si le format du jeu d'instructions le permet.	4
Adresse de l'instruction dans le programme du processeur (@P)	Nous ne pouvons stocker que 12 bits de l'adresse de l'instruction dans le format du jeu d'instructions, pour contourner les problèmes posés par la MMU (Voir paragraphe II.2).	12
Adresse de l'instruction de destination dans le programme du watchdog (@W)	Contrairement au champ précédent, l'adresse de l'instruction de destination dans le programme du watchdog ne sert pas à vérifier si le processeur exécute la bonne instruction. Ce champ doit adresser les instructions watchdog pour pouvoir y sauter dans les cas de jump. Chaque instruction watchdog doit donc avoir une adresse unique. Allouer 20 bits pour ce champ nous permettra d'adresser 2^{20} instructions watchdog. Ce chiffre doit être en théorie suffisant. Si le programme watchdog doit excéder cette taille, il ne pourra pas être géré par IDSM dans cette version.	20
Adresse relative de l'instruction dans la mémoire	Etant donné que nous avons fait la supposition que la taille du bloc ne dépassera pas 15 instructions (création d'un nouveau bloc dans le cas contraire), l'adresse	4

du processeur (@R)	relative dans le bloc ne nécessitera pas plus de 4 bits.	
Flag1	La taille de ce champ est de 1 bit.	1
Flag2	La taille de ce champ est de 1 bit.	1

A.1.2. Répartition des champs

Ces champs seront répartis dans les instructions du watchdog comme suit :

Tableau A-III: Répartition des champs dans le jeu d'instructions

	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
Type	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX
SH	XX	XX	XX	X	X	X	X	X	X	X	X	X	X	X	X	X
SV							XX	XX	XX	XX						
@V		XX	XX		XX	XX						XX	XX		XX	XX
Taille	XX	XX	XX								XX	XX	XX	XX	XX	XX
@P	XX	XX	XX	XX	XX	XX										
@W							XX	XX	XX	XX						
@R				XX	XX	XX										
Flag1							XX	XX								
Flag2																XX

X : optionnel

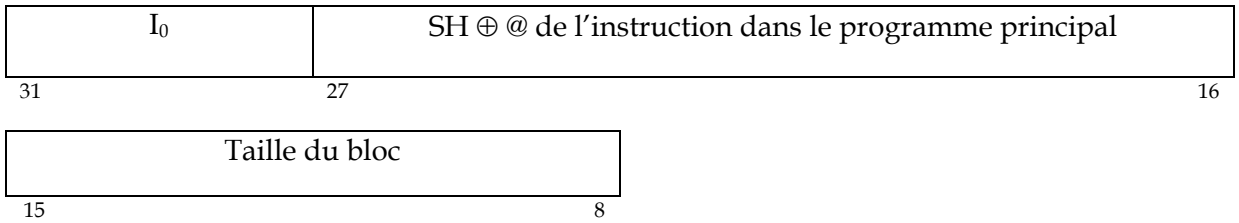
XX : obligatoire

A.2. Description du jeu d'instructions du watchdog

I_0 : Destination

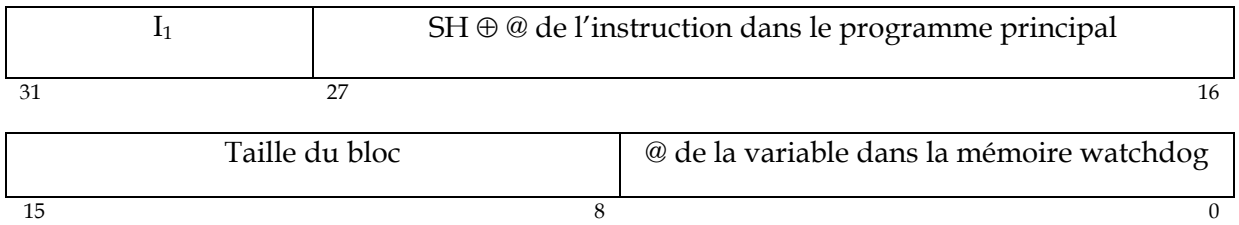
Etant donné que cette instruction est l'instruction la plus récurrente d'après les statistiques effectuées, il vaudrait mieux réduire sa taille au maximum dans l'optique de réduire le surcoût mémoire. Les instructions de type I_0 contiennent un champ taille du bloc pour pouvoir repérer la singularité suivante I_6, I_7 et I_8 (les instructions de type I_3, I_4 et I_5 sont repérées différemment).

Dans ce format d'instruction, la SH de référence est hachée avec les 4 bits de poids plus faible de l'adresse de l'instruction dans le programme principal. Sachant qu'il faut comparer cette adresse avec celle prise par le processeur, on procède comme suit : on dispose déjà de l'instruction à compacter (venant du processeur), on envoie donc cette instruction dans le bloc de calcul de signatures (les bascules ayant été initialisées à 0). Une fois la SH calculée, nous pourrons déduire l'adresse de l'instruction dans le programme principal. Celle-ci est comparée à l'adresse du processeur. Outre le calcul et la comparaison de l'adresse, il faut également mémoriser la taille. Il faudra donc 1 seul cycle pour exécuter cette instruction.



I_1 : Destination LOAD

Cette instruction est identique à la précédente, excepté le fait qu'on doive également comparer une donnée du processeur avec une variable présente dans la mémoire du watchdog. La demande de lecture se fait dans le même cycle que la mémorisation de la taille du bloc. On reçoit également la donnée de la mémoire du watchdog avant le début du cycle suivant (le temps d'accès étant inférieur à un cycle). Il est alors possible de la comparer. Il faudra tout de même attendre un peu pour avoir un résultat de comparaison véritablement significatif, le temps que les signaux soient stables.



I_2 : Destination STORE

Pour I_2, on mémorise la taille du bloc en un cycle, et on récupère la donnée à écrire pendant ce même cycle, pour l'envoyer vers le module d'interface mémoire, avec l'adresse à laquelle cette donnée doit être stockée. Au cycle suivant, l'interface mémoire prendra en compte la demande d'écriture, ainsi que les données nécessaires.

I_2	SH ⊕ @ de l'instruction dans le programme principal

31 27 16

Taille du bloc	@ de la variable dans la mémoire watchdog

15 8 0

I_3 : Instruction fréquente

Cette instruction, comme I_4 et I_5, est repérée par le champ adresse relative de l'instruction dans le programme principal. Pour chaque instruction fréquente, le watchdog la compacte et compare la signature obtenue avec une signature horizontale. Cela peut se faire en un seul cycle, étant donné que tous les signaux sont disponibles (il faudra cependant attendre le cycle suivant pour avoir un signal d'erreur stable). Un minimum d'un cycle est nécessaire.

I_3	Sign. horizontale	@ de l'inst **relative** dans le prog. processeur

31 27 23 16

I_4 : Instruction LOAD

De la même manière qu'à chaque fois qu'il est nécessaire de lire une donnée en mémoire pour la comparer ensuite, on va extraire la variable critique via le module d'interface processeur au premier cycle, en même temps qu'on demandera à l'interface mémoire de lire la donnée à l'adresse qu'on lui aura indiqué. Il est également nécessaire d'effectuer la comparaison entre la variable critique du processeur, et la variable stockée dans la mémoire du watchdog, ainsi que la comparaison entre la signature de référence, et la signature calculée, puisque les signaux sont disponibles, ou en train d'être calculés. Il suffira d'attendre le cycle suivant pour obtenir les résultats de comparaison qu'il faudra prendre en compte. Un seul cycle est donc nécessaire pour exécuter cette instruction watchdog.

I_4	Sign. horizontale	@ de l'inst **relative** dans le prog. processeur

31 27 23 16

	@ de la variable dans la mémoire watchdog

15 8 0

I_5 : Instruction STORE

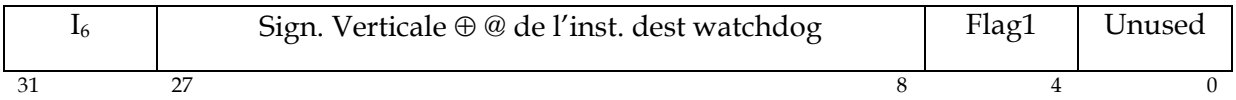
Au cours du premier cycle d'exécution de cette instruction, on calcule la signature horizontale, et on extrait la variable critique à écrire dans la mémoire watchdog. Dans le même cycle, on peut comparer la signature horizontale et envoyer l'adresse extraite à l'interface mémoire. Il faudra cependant attendre le front montant suivant pour que le signal indiquant le résultat de la comparaison soit stable, et que le module interface mémoire prenne correctement en compte les informations à écrire dans la mémoire du watchdog. Un seul cycle est alors nécessaire au module exécuteur pour effectuer ces actions.

I_6 : Branchement conditionnel

Lorsque le processeur arrive sur une instruction de branchement conditionnel, le watchdog devra attendre pour voir si le processeur va effectuer le branchement ou pas :

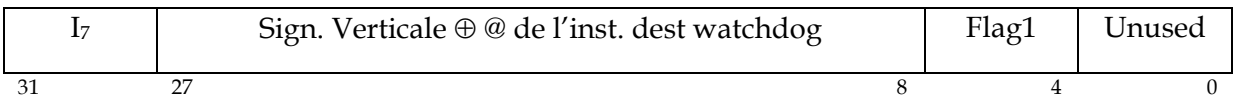
- Si le branchement est effectué, le watchdog charge le nœud de destination, et une fois ce nœud chargé, il compare son adresse avec celle du nœud pris par le processeur.
- Si le branchement n'est pas effectué, le watchdog charge le nœud situé en séquence dans son programme de vérification et attend que le processeur arrive sur une autre singularité.

Il y a donc 2 adresses possibles pour l'instruction suivante: (1) celle qui suit immédiatement l'instruction de branchement et (2) celle hachée avec la signature verticale et à laquelle le watchdog devra sauter dans le cas où le branchement est pris Il faut donc calculer la signature verticale, pour pouvoir extraire l'adresse de l'instruction watchdog. Cette dernière doit évidemment correspondre à un début de bloc.



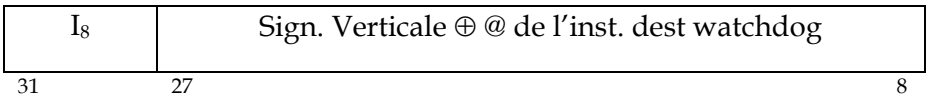
I_7 : Branchement inconditionnel

Cette instruction revient donc à exécuter l'instruction I_6 dans le cas où le branchement est pris. Il faut donc calculer la signature verticale pour pouvoir extraire l'adresse de l'instruction de destination dans le programme du watchdog. Un minimum d'un cycle est alors nécessaire pour pouvoir effectuer cette instruction.



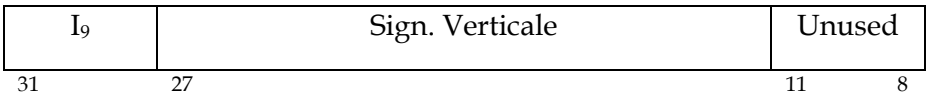
I_8 : Saut vers un sous programme

Cette instruction correspond à un branchement inconditionnel, où une sauvegarde de la signature et de l'adresse est effectuée. De la même manière que dans I_7, le watchdog calcule sa signature verticale et extrait l'adresse de la fonction appelée dans le programme du watchdog. Ensuite, il remet à zéro le compacteur pour pouvoir vérifier le flot de contrôle de la fonction de manière disjointe.



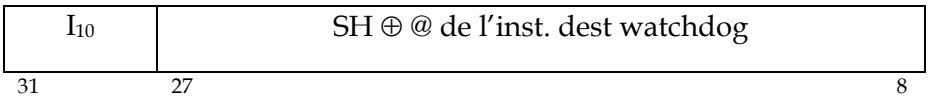
I_9 : Retour d'un sous programme

Cette instruction ressemble à un branchement inconditionnel sauf que l'adresse de destination est stockée dans la pile. De plus, une fois que la signature verticale est calculée, il faut charger le MISR avec une signature enregistrée dans la pile. Pour cela, au premier cycle, on demande à la comparer avec la référence (même si le résultat ne sera stable qu'au cycle suivant). En même temps, on extrait de la pile la signature à charger dans le MISR. Au cycle suivant, lorsqu'une nouvelle donnée arrive du processeur principal, on envoie la signature à charger dans le MISR, pour qu'elle fasse office de valeur initiale pour le calcul de la signature suivante (il s'agira également d'exécuter une instruction de destination). Un minimum de deux cycles est alors nécessaire.



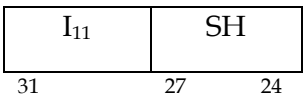
I_{10} : Instruction de saut en dehors du programme principal

Cette instruction correspond à un branchement conditionnel. Le watchdog calcule d'abord une signature horizontale et ensuite l'adresse de l'instruction dans sa mémoire. Le watchdog n'effectuera le saut que si le processeur le fait.



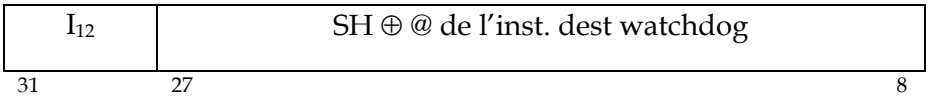
I_{11} : Instruction de non saut en dehors du programme principal

Le watchdog calcule la signature horizontale et la compare à la signature de référence.



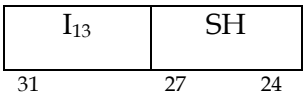
I_{12} : Saut vers un sous programme en dehors du programme principal

Cette instruction correspond également à un branchement inconditionnel. De la même manière que dans I_8, le watchdog calcule sa signature verticale et extrait l'adresse de la fonction appelée dans le programme du watchdog. Le watchdog doit également sauvegarder l'adresse de l'instruction watchdog appelante pour pouvoir y revenir à la fin du sous programme. Ici, il n'est pas nécessaire de sauvegarder la signature intermédiaire, vu qu'il n'y a pas de contrôle sur des signatures verticales pour les instructions en dehors du programme principal.



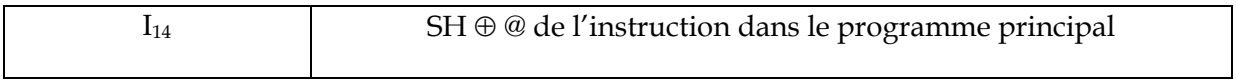
I_{13} : Retour d'un sous programme en dehors du programme principal

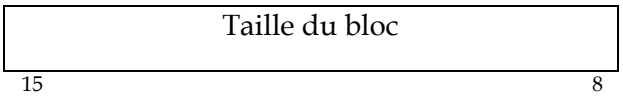
Le watchdog calcule d'abord la signature horizontale et la compare à la signature de référence. Ensuite le watchdog fera un saut vers l'adresse de l'instruction sauvegardée au moment du call.



I_{14} : Instruction Save

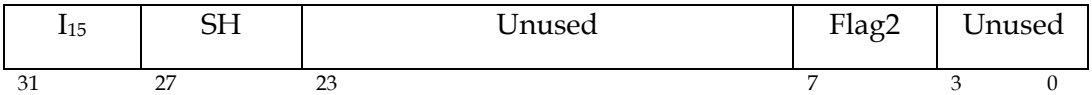
Pendant cette instruction le watchdog décrémente son pointeur sur sa pile. Etant donné que cette instruction peut aussi être une instruction de destination, elle possède le même format que l'instruction I_0.





I_{15} : Instruction Restore

Pendant cette instruction, le watchdog incrémente son pointeur sur sa pile. Si cette instruction correspond aussi à une instruction de destination son exécution dépendra de l'instruction précédente. En effet, si cette dernière est une instruction de branchement, le watchdog ne procèdera pas à la vérification de la signature horizontale ainsi que celle verticale de l'instruction de I_9 suivante.



A.3. Enchaînement possible des instructions watchdog

Nous recensons dans le Tableau A-IV les différentes possibilités de séquences pour les instructions watchdog.

Tableau A-IV: Enchaînement fonctionnel possible des instructions watchdog

Instruction courante	Instruction suivante possible
I_0	I_3, I_4, I_5, I_6, I_7, I_8
I_1	I_3, I_4, I_5, I_6, I_7, I_8, I_9
I_2	I_3, I_4, I_5, I_6, I_7, I_8, I_9
I_3	I_3, I_4, I_5, I_6, I_7, I_8, I_9
I_4	I_3, I_4, I_5, I_6, I_7, I_8, I_9
I_5	I_3, I_4, I_5, I_6, I_7, I_8, I_9
I_6	I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9
I_7	I_0, I_1, I_2
I_8	I_0, I_1, I_2
I_9	I_0, I_1, I_2
I_{10}	I_0, I_1, I_2, I_{10}, I_{11}, I_{12}, I_{13}
I_{11}	I_{10}, I_{11}, I_{12}, I_{13}
I_{12}	I_0, I_1, I_2, I_{10}, I_{11}, I_{12}, I_{13}
I_{13}	I_{10}, I_{11}, I_{12}, I_{13}
I_{14}	I_3, I_4, I_5, I_6, I_7, I_8 ,I_{10}, I_{11}, I_{12}, I_{13}
I_{15}	I_3, I_4, I_5, I_6, I_7, I_8 ,I_{10}, I_{11}, I_{12}, I_{13}

A.4. Organisation de la mémoire du watchdog

La mémoire du watchdog est physiquement séparée de la mémoire du processeur principal, et peut être organisée différemment. Voici les différents schémas possibles :

- Une instruction / ligne (la ligne est alors de 32 bits) : c'est la méthode la plus facile à mettre en œuvre mais elle induit des coûts mémoire assez importants à cause de nombreux bits inutilisés.
- Plusieurs instructions / ligne : Ce schéma peut réduire les surcoûts mémoire mais nécessite l'ajout d'un module dans le watchdog qui s'occupe du pré-fetch. Nous proposons 3 schémas d'organisation impliquant plusieurs instructions par ligne :
  - Tous les débuts de blocs sont au début de la ligne, mais on risque de se retrouver souvent dans le cas d'une seule instruction par ligne.
  - Adressage par octet qui offre une gestion assez facile des accès mémoire mais nous oblige à sacrifier 2 bits dans le champ « @ de l'inst. dans le prog. watchdog »
  - Solution mixte : Adressage par 2 octets et chaque début de bloc est soit au début soit au deuxième octet de la ligne. Dans cette solution, un seul octet est perdu pour chaque début de bloc au maximum, et un seul bit est sacrifié dans le champ « @ de l'inst. dans le prog. watchdog »

Pour notre prototype, nous avons opté pour le premier schéma à savoir une instruction par ligne, notamment pour limiter les pertes de cycles pouvant être induites par des problèmes d'alignement des instructions dans les autres cas. Une optimisation de cette organisation peut être envisagée pour des travaux ultérieurs.

Annexe -B-

Développement des outils pour la génération du programme du watchdog

Pour que notre méthode de vérification de flot de contrôle soit la plus portable possible, il faut que le programme du watchdog soit généré indépendamment du langage source de l'application à vérifier et de l'architecture de la machine cible. Ceci peut être fait pendant la phase de compilation de l'application : le compilateur permet à la fois de générer le fichier exécutable pour le processeur et le programme de vérification pour le watchdog. De cette façon, tout le mécanisme de génération du programme du watchdog est transparent pour le programmeur.

Cependant, comme nous l'avons expliqué dans le paragraphe III.2, la génération du programme du watchdog au cours de la phase de compilation (et donc avant l'édition des liens) ne tient compte ni des appels à des fonctions des bibliothèques ni des appels système. Ainsi, tout le code inséré dans le programme après l'étape d'édition des liens ne peut pas être vérifié.

En outre, les adresses obtenues pendant la phase de compilation sont toutes relatives au début du main. En d'autres termes, si le programme inclut des appels système et des appels à des bibliothèques, les adresses calculées pendant la compilation ne sont pas les adresses utilisées au cours de l'exécution du programme. La conséquence directe de ces différences dans les adresses est la génération de signatures erronées.

Par conséquent, la génération du programme du watchdog doit être décomposée en deux étapes : pré-édition des liens et post-édition des liens:

- Pré-édition des liens : utilise une version modifiée de GCC-4.4.2. Le choix de la suite de développement logiciel standard GNU-Linux, dont notamment GCC, est surtout motivé par des raisons de licence et de portabilité.

- Post-édition des liens: la seule manière d'explorer le fichier binaire généré après l'édition des liens est de parser son fichier de désassemblage. Ce fichier contient non seulement le code assembleur, mais aussi les instructions en code binaire correspondant qui peut être utilisé pour la génération des signatures.

Cette annexe décrit le mécanisme de génération du programme du watchdog. Le paragraphe B.1 est consacré à la phase de pré-édition des liens et à la modification de GCC, tandis que le paragraphe B.2 décrit la phase post-édition des liens.

B.1. Etapes pré-édition des liens pour la génération du programme du watchdog

Comme illustré dans la figure B-1, l'entrée de cette étape est le code source de l'application à vérifier, et sa sortie est le squelette du programme watchdog de la forme :

```
@processeur : Instruction watchdog préliminaire.
```

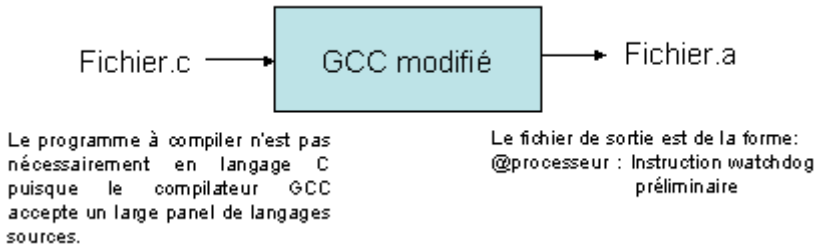


Figure B-1 : Entrée et sortie de la 1ère étape de la génération du programme du watchdog

La plupart des étapes de compilation se fait sur une représentation intermédiaire appelée Register Transfert Language (RTL). RTL est inspiré par les listes LISP et est indépendant du langage de programmation et de l'architecture cible. Afin de générer le programme de surveillance, nous analysons le code RTL pour identifier les singularités et leurs adresses. Les modifications de GCC comprennent l'analyse de la criticité des variables et des registres.

Localisation des singularités

La représentation du code RTL pour une fonction est une chaîne doublement chaînée d'objets appelés insns. Les insns peuvent être de type INSN, JUMP_INSN, BARRIER, LABEL, CALL_INSN ou bien NOTE.

```
Si (CODE_INSN(insn)==JUMP_INSN)
       Si (RTX_LABEL(insn) ==NULL) //"Retour d'un sous programme"
       Sinon si (CODE_INSN(NEXT_INSN(insn))==BARRIER)// "Branchement inconditionnel"
       Sinon //"Branchement conditionnel"
Sinon si (CODE_INSN(insn)==CALL_INSN) //"Branchement vers un sous programme"
Sinon si (BB_HEAD(BASIC_BLOCK(bb_number))==insn) // "destination"
```

Identification des variables critiques

Trois critères de criticité ont été définis pour identifier les variables critiques : durée de vie, dépendances fonctionnelles et participation dans les conditions de branchement. Ces critères de criticité sont calculés automatiquement lors de la compilation avec la version modifiée de GCC.

Un seuil de criticité est défini par l'utilisateur afin que seuls les registres critiques soient vérifiés. De cette façon, un compromis peut être défini entre l'exhaustivité de la vérification de l'intégrité des données et la taille de la mémoire du watchdog.

La durée de vie d'une variable ou d'un registre est un paramètre très important de criticité. Nous ne proposons pas de nouvelle technique pour la calculer, mais nous profitons des analyses précises existant dans les compilateurs modernes. Dans notre cas, nous avons utilisé les fonctions et les structures de données existant dans GCC. Au niveau instruction, nous avons utilisé la macro REG_LIVE_LENGHT.

Localisation des instructions fréquentes

GCC gère les informations de profil d'exécution de chaque programme compilé, dont notamment la fréquence d'exécution. Cette fréquence est une estimation de la fréquence d'exécution du bloc dans une fonction. Il est représenté par un entier variant entre 0 et BB_FREQ_BASE. La fréquence du bloc le plus souvent exécuté dans une fonction donnée est initialement fixée à BB_FREQ_BASE et les autres fréquences sont ajustées en conséquence.

Pour localiser les instructions fréquentes, nous profitons de ces informations sur le profil. Pour chaque bloc linéaire on calcule un quotient de fréquences avec l'équation suivante :

$$Frequency\ (BB) = BB\ ->\ frequency * 100 / BB\_FREQ\_BASE$$

B.1.1. Première phase- repérage des singularités

La première phase de l'étape pré-édition des liens consiste à parcourir le programme RTL pour récolter des informations sur les singularités telles que le type et l'adresse. Cette phase est illustrée dans la figure B-2.

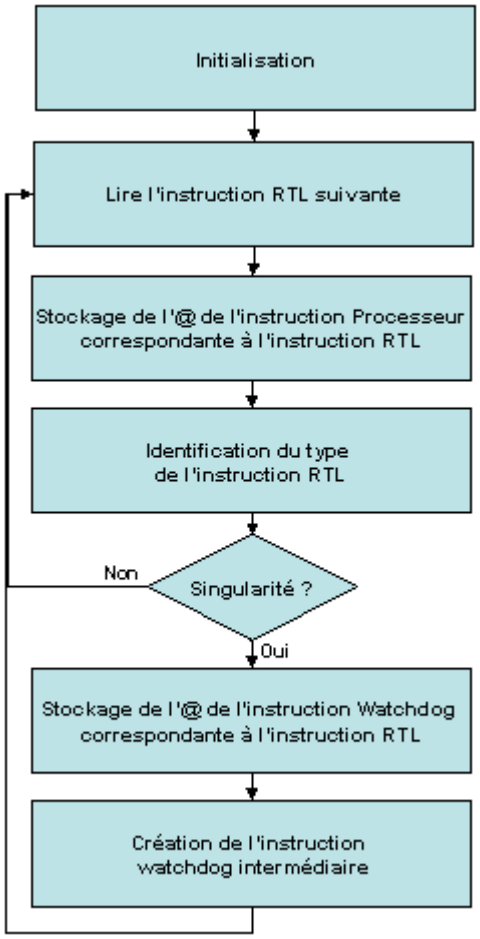


Figure B-2 : Phase de repérage des singularités

A la fin de cette phase nous disposons de 3 listes :

- Liste des instructions watchdog intermédiaires
- Liste des adresses des instructions processeur pour chaque instruction RTL
- Liste des adresses des instructions watchdog pour chaque instruction RTL

Toutes les informations nécessaires à la génération du programme du watchdog ont donc été récoltées pendant cette première phase. Il suffit de les rassembler durant la 2ème phase.

B.1.2. Deuxième phase – génération d'un programme préliminaire

Pendant cette phase nous tentons de rassembler les informations collectées lors de la première phase pour pouvoir générer un programme watchdog préliminaire. En effet, à la fin de cette phase nous serons en mesure de faire le lien entre chaque instruction de saut avec son instruction de destination. Cette phase est illustrée dans la figure B-3.

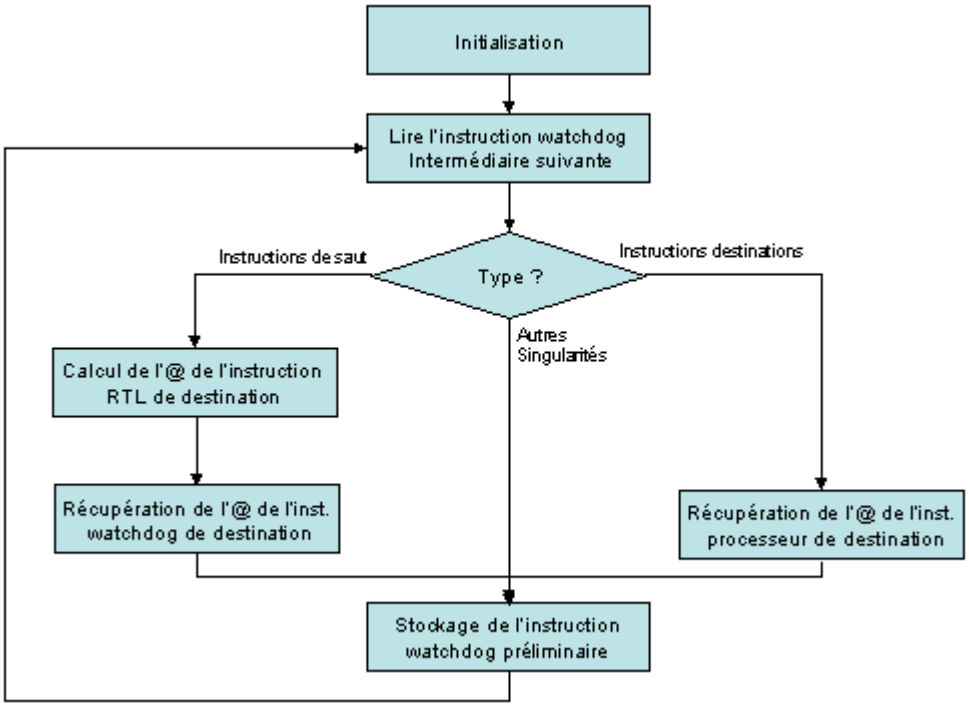


Figure B-3 : Génération d'un programme préliminaire

B.2. Etape post-édition des liens pour la génération du programme du watchdog

Pour explorer le fichier exécutable généré après l'édition des liens, on parse son fichier de désassemblage obtenu avec les commandes suivantes :

```
$sparc-elf-objdump -d Fichier.o  > Fichier.s
```

Ce fichier contient non seulement le code assembleur, mais aussi les codes binaires correspondant. Ces derniers peuvent être utilisés pour la génération des signatures de référence.

A partir de ce fichier et du programme préliminaire, nous pouvons générer le programme du watchdog complet (voir la figure B-4).

Figure B-4 : Entrées et sortie de la 2ème étape de la génération du programme du watchdog

Pour toutes les instructions ajoutées après la phase d'édition des liens, le watchdog calcule une signature horizontale. De plus, si l'instruction est une instruction de saut, le watchdog doit être capable de faire un saut vers l'instruction de destination correspondante dans son programme.

Les informations à récolter sont donc :

- Pour les instructions de saut
  - Type
  - Signature horizontale
  - L'adresse de la destination dans le programme de watchdog.
- Pour les autres instructions
  - Type
  - Signature horizontale

Le parsing du fichier de désassemblage doit se fait ligne par ligne. Cependant, dans la première lecture, l'adresse de l'instruction destination dans le programme du watchdog, nécessaire pour les instructions de saut, peut ne pas être disponible (si l'instruction correspondante dans le programme principal n'a pas encore été parsée). D'où la nécessité de faire l'étape post-link en 2 phases également.

B.2.1. Première phase - parsing du fichier assembleur

La description de cette phase est détaillée dans la figure B-5. Elle consiste en gros à parcourir le fichier de désassemblage pour récolter les informations nécessaires pour la génération du programme du watchdog. Voici le format du fichier de désassemblage :

```
Disassembly of section .text:

00000000 <_text_start>:
       0:    88 10 00 00  mov  %g0, %g4
       4:    09 00 00 04  sethi  %hi(0x1000), %g4
       8:    81 c1 23 70  jmp  %g4 + 0x370    ! 1370 <_hardreset_mvt>
       c:    01 00 00 00  nop
[…]
000014bc <main>:
    14bc:    9d e3 bf 98  save  %sp, -104, %sp
    14c0:    2b 00 00 07  sethi  %hi(0x1c00), %l5
 […]
```

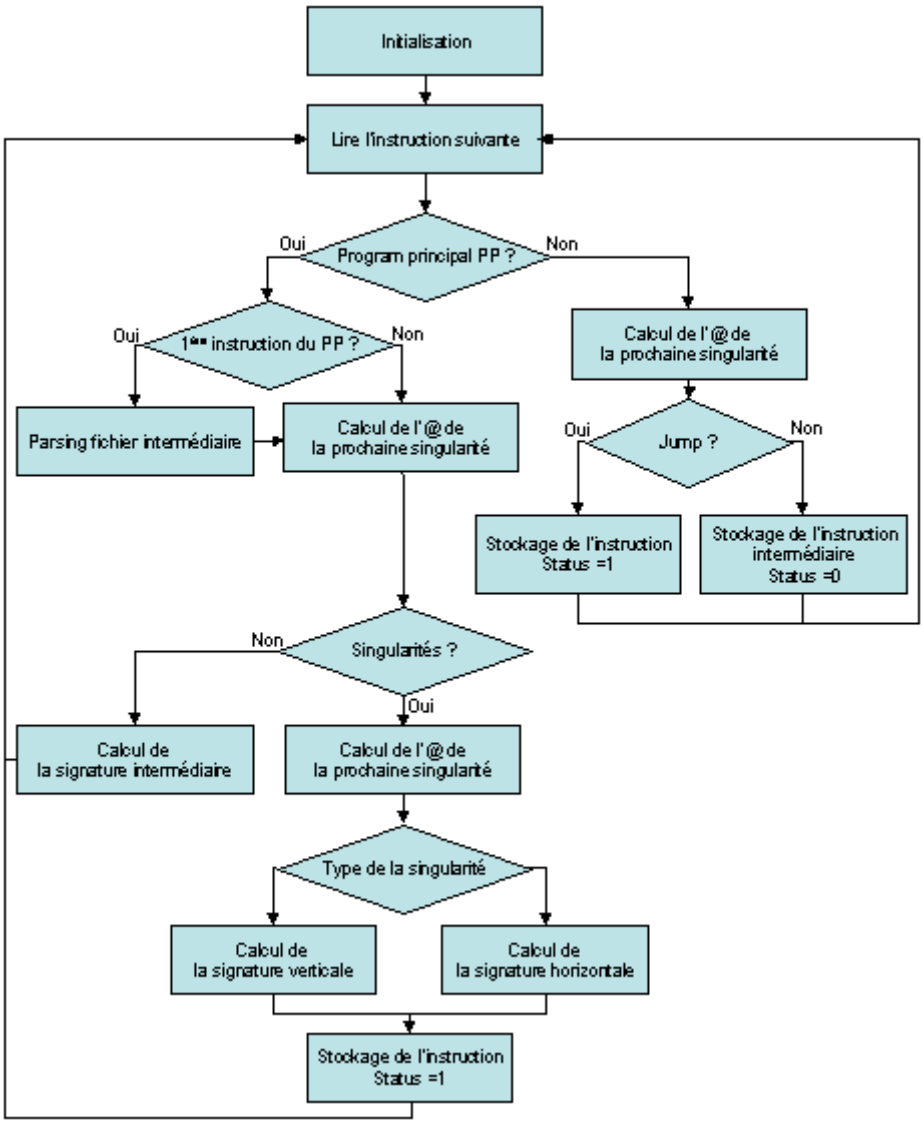


Figure B-5 : Parsing du fichier assembleur

A la lecture de la 1ère instruction appartenant au programme principal, le parsing du fichier.a (généré pendant la phase de pré-link) est fait.

Pour identifier les instructions de saut, nous localisons les mnémoniques de saut spécifiques à l'architecture cible. Ces mnémoniques sont sauvegardées dans un fichier de configuration. Pour le cas du Sparc v8, la liste des mnémoniques est: {BA, BN, BNE, BE, BG, BLE, BGE, BL, BGU, BLEU, BCC, BCS, BPOS, BNEG, BVC, BVS}.

Status est une liste qui indique pour chaque instruction watchdog si elle est achevée ou pas.

B.2.2. Deuxième phase – Génération du programme final du watchdog

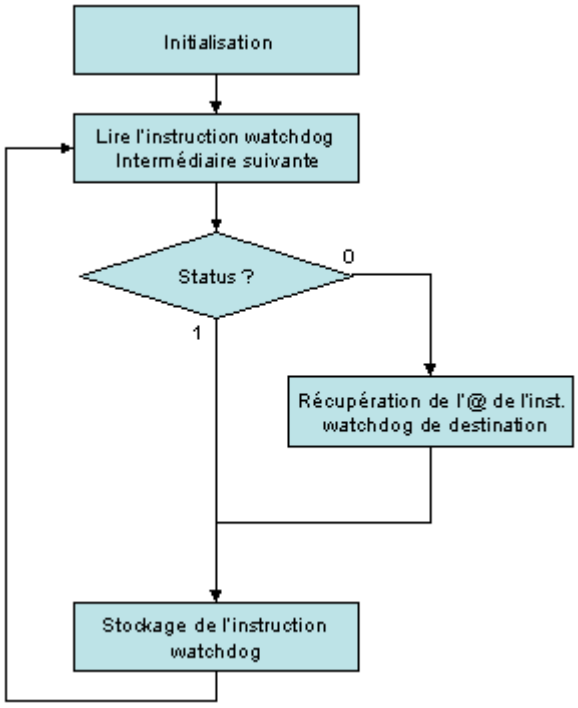


Figure B-6 : Génération du programme final du watchdog

Pendant cette deuxième phase on parcourt la liste des instructions watchdog intermédiaires tout en complétant les instructions incomplètes (status = 0) avec l'adresse de destination dans le programme du watchdog (voir la figure B-6).

Le retour d'un traitement d'exception est un cas particulier de saut, car l'adresse de retour n'est pas connue à l'avance. Sa valeur dans le programme de génération est égale à -1. La reconnaissance du retour de la routine d'exception ne pose aucun problème ; le watchdog devra recharger le dispositif de compaction avec la signature empilée pendant le départ vers la routine d'exception. Le watchdog devra également restaurer son pointeur de programme et charger le nœud qui lui correspond dans son application de vérification.

Annexe -C-

Evaluation de la criticité en fonction des options d'optimisation

Cette annexe présente les résultats complets des variations des critères de criticité en fonction des options d'optimisation employées pendant la compilation d'un programme : durée de vie (tableau C-I), fanout (tableau C-II), participation aux sauts (tableau C-III) et dépendances fonctionnelles (tableau C-IV), et ce pour plusieurs applications du benchmark mibench.

Tableau C-I: Variations du critère "durée de vie" par rapport aux fonctions d'optimisation

	adpcm	basicmath	blowfish	CRC32	FFT	gsm	Jpeg	patricia	qsort	AES	stringsearch	Moyenne	Variance
fcse-follow-jumps	100	100	99,76	100	100	99,22	99,92	99,66	100,8	101,28	99,63	100	0,32
fcse-skip-blocks	99,44	99,37	101,59	100	100	100,6	99,43	100,04	100,8	101,44	100,24	100,3	0,59
fexpensive-optimizations	100	100	100	100	100	99,97	99,99	100	100	100	100	99,99	0
fgcse	104,7	196,08	110,54	100	103,7	108,1	100,7	103,08	113,8	136,11	100,45	116,1	809,16
finline-functions	100	100	100,39	100	99,2	138,7	127,0	177,53	100	105,91	123,97	115,7	613,85
foptimize-sibling-calls	100	100	100	100	100	99,59	99,32	100	100	100	99,98	99,89	0,05
fregmove	100	100	99,96	100	100	99,29	99,97	100	100	100	99,31	99,86	0,07
frerun-cse-after-loop	106,4	112,25	105,77	101,48	100,3	100,8	102	104,34	104,2	99,79	102,95	103,7	12,9
fschedule-insns2	100	100	100	100	100	99,75	99,92	100	100	100	100	99,97	0,01
fschedule-insns	99,03	102,23	103,69	96,61	99,6	104,8	99,79	98,95	101,2	129,96	98,88	103,2	84,57
fstrict-overflow	100	100	102,68	100	100	100,2	100	100	100	100,35	102,46	100,5	1,04
fthread-jumps	100	100	100	100	100	100	99,97	100	100	100	100	99,99	0
ftree-vrp	98,37	102,79	104,9	100	100	100	100,7	100	107,4	100,46	102,4	101,5	6,98
ftree-pre	100,9	100	100,75	92,7	98	102,1	101,8	104,26	100	100	101,58	100,2	8,63
fcaller-saves	100	100	100	100	100	100	100	100	100	100	100	100	0
funswitch-loops	100	100	100	100	102,8	100	100,3	100	100	100	99,06	100,2	0,85
O2	108,7	204,24	129,77	92,81	101,4	113,9	100,5	108,06	111,5	153,39	101,07	120,5	1050,8
O3	108,7	204,24	145,17	92,81	103,3	276,6	138,1	180,11	111,5	216,48	118,51	154,1	3391,3
Os	103,2	109,98	115,52	92,28	98,26	107	89,19	98,97	83,05	118,52	93,77	100,9	124,06

Tableau C-II: Variations du critère "Fanout " par rapport aux fonctions d'optimisation

	adpcm	basicmath	blowfish	CRC32	FFT	gsm	jpeg	patricia	qsort	AES	string	Moyenne	Variance
fcse-follow-jumps	**100**	**100**	**98,98**	**100**	**100**	**100,2**	**100**	**104,29**	**106,7**	**97,48**	**99,22**	**100,6**	**6,65**
fcse-skip-blocks	**98,63**	**115,13**	**100**	**100**	**100**	**101,2**	**100,1**	**103,68**	**106,7**	**98,2**	**99,22**	**102,1**	**24,66**
Fexpensive-opt	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**
fgcse	**99,31**	**98,91**	**100,76**	**100**	**102,1**	**103,7**	**100**	**108,58**	**100**	**100**	**96,89**	**100,9**	**9,39**
finline-functions	**100**	**100**	**100,5**	**100**	**97,22**	**106,4**	**103,6**	**116,56**	**100**	**107,55**	**108,15**	**103,6**	**31,44**
foptimize-sibling-calls	**100**	**100**	**100**	**100**	**100**	**99,35**	**100**	**100**	**100**	**100**	**100**	**99,94**	**0,038**
fregmove	**100**	**100**	**100**	**100**	**100**	**100,2**	**99,98**	**100**	**100**	**100**	**99,22**	**99,94**	**0,063**
frerun-cse-after-loop	**100**	**102,16**	**100**	**100**	**100**	**99,83**	**99,89**	**100**	**100**	**96,4**	**100,97**	**99,93**	**1,85**
fschedule-insns2	**101,36**	**98,91**	**112,21**	**107,89**	**99,3**	**103,9**	**110**	**106,74**	**102,7**	**125,17**	**108,15**	**106,9**	**55,31**
fschedule-insns	**98,63**	**100**	**100,25**	**105,26**	**102,8**	**108,4**	**102,5**	**99,38**	**102,7**	**193,16**	**99,22**	**110,2**	**765,63**
fstrict-overflow	**100**	**100**	**101,52**	**100**	**100**	**100,6**	**100,4**	**100**	**100**	**101,79**	**104,27**	**100,8**	**1,75**
fthread-jumps	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**
ftree-vrp	**100,68**	**102,16**	**107,12**	**100**	**100**	**99,75**	**100,2**	**100**	**110,7**	**100,71**	**100**	**101,9**	**12,90**
ftree-pre	**102,04**	**100**	**102,79**	**102,63**	**99,3**	**102,5**	**101,6**	**106,13**	**100**	**100**	**101,16**	**101,7**	**3,71**
fcaller-saves	**100**	**101,08**	**100,5**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100,1**	**0,11**
funswitch-loops	**100**	**100**	**100**	**100**	**100,7**	**100**	**100,2**	**100**	**100**	**100**	**99,22**	**100**	**0,11**
O2	**104,08**	**121,62**	**116,53**	**113,15**	**106,3**	**116,6**	**111,9**	**111,04**	**85,33**	**210,43**	**105,82**	**118,4**	**1021,4**
O3	**104,08**	**121,62**	**116,79**	**113,15**	**104,2**	**125,1**	**115,9**	**122,69**	**85,33**	**226,97**	**110,48**	**122,4**	**1328,4**
Os	**100**	**130,27**	**112,21**	**105,26**	**99,3**	**104,3**	**99,28**	**96,31**	**81,33**	**196,04**	**97,47**	**111,1**	**935,62**

Tableau C-III: Variations du critère "Participation aux sauts" par rapport aux fonctions d'optimisation

	adpcm	basicmath	blowfish	CRC32	FFT	gsm	jpeg	patricia	qsort	AES	string	Moyenne	Variance
fcse-follow-jumps	**100**	**100**	**99,19**	**100**	**100**	**100**	**99,99**	**103,03**	**105,5**	**97,97**	**99,09**	**100,4**	**4,28**
fcse-skip-blocks	**98,9**	**111,86**	**100**	**100**	**100**	**100,8**	**100,1**	**102,59**	**105,5**	**98,55**	**99,39**	**101,6**	**15,34**
fexpensive-opt	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**0**
fgcse	**100,54**	**99,15**	**100,6**	**100**	**101,6**	**102,5**	**100,0**	**106,06**	**100**	**100**	**97,58**	**100,7**	**4,68**
finline-functions	**100**	**100**	**100,4**	**100**	**98,44**	**114,3**	**111,3**	**133,33**	**100**	**107,24**	**109,05**	**106,7**	**108,12**
foptimize-sibling-calls	**100**	**100**	**100**	**100**	**100**	**99,56**	**100**	**100**	**100**	**100**	**100**	**99,96**	**0,01**
fregmove	**100**	**100**	**100**	**100**	**100**	**100,2**	**99,99**	**100**	**100**	**100**	**99,39**	**99,95**	**0,03**
frerun-cse-after-loop	**100**	**101,69**	**100**	**100**	**100**	**99,89**	**99,92**	**100**	**100**	**97,1**	**100,75**	**99,94**	**1,18**
fschedule-insns2	**101,09**	**99,15**	**109,63**	**106,25**	**99,48**	**102,7**	**107,1**	**104,76**	**102,2**	**120,29**	**106,33**	**105,4**	**35,47**
fschedule-insns	**98,9**	**100**	**100,2**	**104,16**	**102,1**	**105,7**	**101,7**	**99,56**	**102,2**	**175,07**	**99,39**	**108,1**	**497,92**
fstrict-overflow	**100**	**100**	**101,2**	**100**	**100**	**100,4**	**100,3**	**100**	**100**	**101,44**	**103,31**	**100,6**	**1,07**
fthread-jumps	**100**	**100**	**100**	**100**	**100**	**100**	**99,98**	**100**	**100**	**100**	**100**	**99,99**	**0**
ftree-vrp	**101,09**	**101,69**	**104,61**	**100**	**100**	**99,07**	**100,1**	**100**	**108,8**	**100,58**	**99,69**	**101,4**	**8,18**
ftree-pre	**101,63**	**100**	**102,2**	**102,08**	**98,96**	**101,4**	**101,2**	**104,32**	**100**	**100**	**100,9**	**101,2**	**2,11**
fcaller-saves	**100**	**100,84**	**100,4**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100,1**	**0,0**
funswitch-loops	**100**	**100**	**100**	**100**	**101,0**	**100**	**100,2**	**100**	**100**	**100**	**99,39**	**100,1**	**0,14**
O2	**103,82**	**116,94**	**112,04**	**110,41**	**104,1**	**110,6**	**108,3**	**107,79**	**87,91**	**188,98**	**104,22**	**114,1**	**670,74**
O3	**103,82**	**116,94**	**110,04**	**110,41**	**103,6**	**186,2**	**120,6**	**137,66**	**87,91**	**206,37**	**110,25**	**126,7**	**1351,7**
Os	**101,09**	**118,64**	**107,83**	**100**	**95,33**	**102,2**	**95,16**	**94,8**	**82,41**	**176,52**	**94,72**	**106,2**	**625,02**

Tableau C-IV: Variations du critère "dépendances fonctionnelles" par rapport aux fonctions d'optimisation

	adpcm	basicmath	blowfish	CRC32	FFT	gsm	jpeg	patricia	qsort	AES	string	Moyenne	Variance
fcse-follow-jumps	100	100	99,19	100	100	100	99,99	103,03	105,5	97,97	99,09	100,4	4,28
fcse-skip-blocks	98,9	111,86	100	100	100	100,8	100,1	102,59	105,5	98,55	99,39	101,6	15,34
fexpensive-opt	100	100	100	100	100	100	100	100	100	100	100	100	0
fgcse	100,54	99,15	100,6	100	101,6	102,5	100	106,06	100	100	97,58	100,7	4,68
finline-functions	100	100	100,4	100	98,44	114,3	111,3	133,33	100	107,24	109,05	106,7	108,12
foptimize-sibling-calls	100	100	100	100	100	99,56	100	100	100	100	100	99,96	0,02
fregmove	100	100	100	100	100	100,2	99,99	100	100	100	99,39	99,95	0,03
frerun-cse-after-loop	100	101,69	100	100	100	99,89	99,92	100	100	97,1	100,75	99,94	1,18
fschedule-insns2	101,09	99,15	109,63	106,25	99,48	102,7	107,1	104,76	102,2	120,29	106,33	105,4	35,47
fschedule-insns	98,9	100	100,2	104,16	102,1	105,7	101,7	99,56	102,2	175,07	99,39	108,1	497,92
fstrict-overflow	100	100	101,2	100	100	100,4	100,3	100	100	101,44	103,31	100,6	1,07
fthread-jumps	100	100	100	100	100	100	99,98	100	100	100	100	99,99	0
ftree-vrp	101,09	101,69	104,61	100	100	99,07	100,1	100	108,8	100,58	99,69	101,4	8,18
ftree-pre	101,63	100	102,2	102,08	98,96	101,4	101,2	104,32	100	100	100,9	101,2	2,11
fcaller-saves	100	100,84	100,4	100	100	100	100	100	100	100	100	100,1	0,07
funswitch-loops	100	100	100	100	101	100	100,2	100	100	100	99,39	100,1	0,14
O2	103,82	116,94	112,04	110,41	104,1	110,6	108,3	107,79	87,91	188,98	104,22	114,1	670,74
O3	103,82	116,94	110,04	110,41	103,6	186,2	120,6	137,66	87,91	206,37	110,25	126,7	1351,7
Os	101,09	118,64	107,83	100	95,33	102,2	95,16	94,8	82,41	176,52	94,72	106,2	625,02

TITRE : Surveillance comportementale de systèmes et logiciels embarqués par signature disjointe

RESUME Les systèmes critiques, parmi lesquels les systèmes embarqués construits autour d'un microprocesseur mono-cœur exécutant un logiciel d'application, ne sont pas à l'abri d'interférences naturelles ou malveillantes qui peuvent provoquer des fautes transitoires. Cette thèse porte sur des protections qui peuvent être implantées pour détecter les effets de telles fautes transitoires sans faire d'hypothèses sur la multiplicité des erreurs générées. De plus, ces erreurs peuvent être soit des erreurs de flot de contrôle soit des erreurs sur les données.

Une nouvelle méthode de vérification de flot de contrôle est tout d'abord proposée. Elle permet de vérifier, sans modifier le système initial, que les instructions du programme d'application sont lues sans erreur et dans le bon ordre. Les erreurs sur les données sont également prises en compte par une extension de la vérification de flot de contrôle. La méthode proposée offre un bon compromis entre les différents surcoûts, le temps de latence de détection et la couverture des erreurs. Les surcoûts peuvent aussi être ajustés aux besoins de l'application. La méthode est mise en œuvre sur un prototype, construit autour d'un microprocesseur Sparc v8.

Les fonctions d'analyse de criticité développées dans le cadre de la méthodologie proposée sont également utilisées pour évaluer l'impact des options de compilation sur la robustesse intrinsèque du logiciel d'application.

Mots clés : Fautes transitoires, vérification de flot de contrôle, watchdog, surveillance à signature disjointe, effet de la compilation sur la robustesse

TITLE: Behavioral monitoring for embedded systems and software by disjoint signature analysis

ABSTRACT Critical systems, including embedded systems built around a single core microprocessor running a software application, can be the target of natural or malicious interferences that may cause transient faults. This work focuses on protections that can be implemented to detect the effects of such transient faults without any assumption about the multiplicity of generated errors. In addition, those errors can be either control flow errors or data errors.

A new control flow checking method is first proposed. It monitors, without modifying the original system, that the instructions of the microprocessor application program are read without error and in the proper order. Data errors are also taken into account by an extension of the control flow checking. The proposed method offers a good compromise between overheads, latency detection and errors coverage. Trade-offs can also be tuned according to the application constraints. The methodology is demonstrated on a prototype built around a Sparc v8 microprocessor.

Criticality evaluation functions developed in the frame of the proposed methodology are also used to evaluate the impact of compilation options on the intrinsic robustness of the application software.

Keywords: transient faults, control flow checking, watchdog processor, disjoint signature monitoring, compilation effect on robustness.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.